wxWidgets 跨平台 GUI 编程

wizardforcel

Published with GitBook



Table of Contents

wxWidgets 跨平台 GUI 编程

来源：Cross-Platform GUI Programming with wxWidgets

作者：Wesley

翻译：未知

第一章 介绍

在这一章中，我们会回答这样一些基本的问题：wxWidgets是什么，它和别的类似的开发库有什么不同。我们还会大概说一下这个项目的历史，以及wxWidgets社区的工作，它采用的许可协议，它的体系架构以及目前拥有的各种版本等。

什么是wxWidgets

wxWidgets是一个程序员的开发工具包，这个工具包用来开发用于桌面或者移动设备的图形界面应用程序。或者说它提供了一个框架，它作了很多底层的管家婆似的工作以便给应用程序提供一些默认的行为。wxWidgets库给程序员提供了大量的类以及类的方法，以供其使用和定制。一个典型图形界面应用程序所作的事情包括：显示一个包含各种标准控件的窗口，也可能需要在窗口中绘制某种特别的图形或者图像，并且还要响应来自鼠标，键盘以及其它输入设备的输入。很可能这个应用程序还要和其它的进程通信，甚至还要驱动别的应用程序，换句话说，wxWidgets可以让程序员编写一个拥有所有通用特性的时髦应用程序的工作变的相对容易。

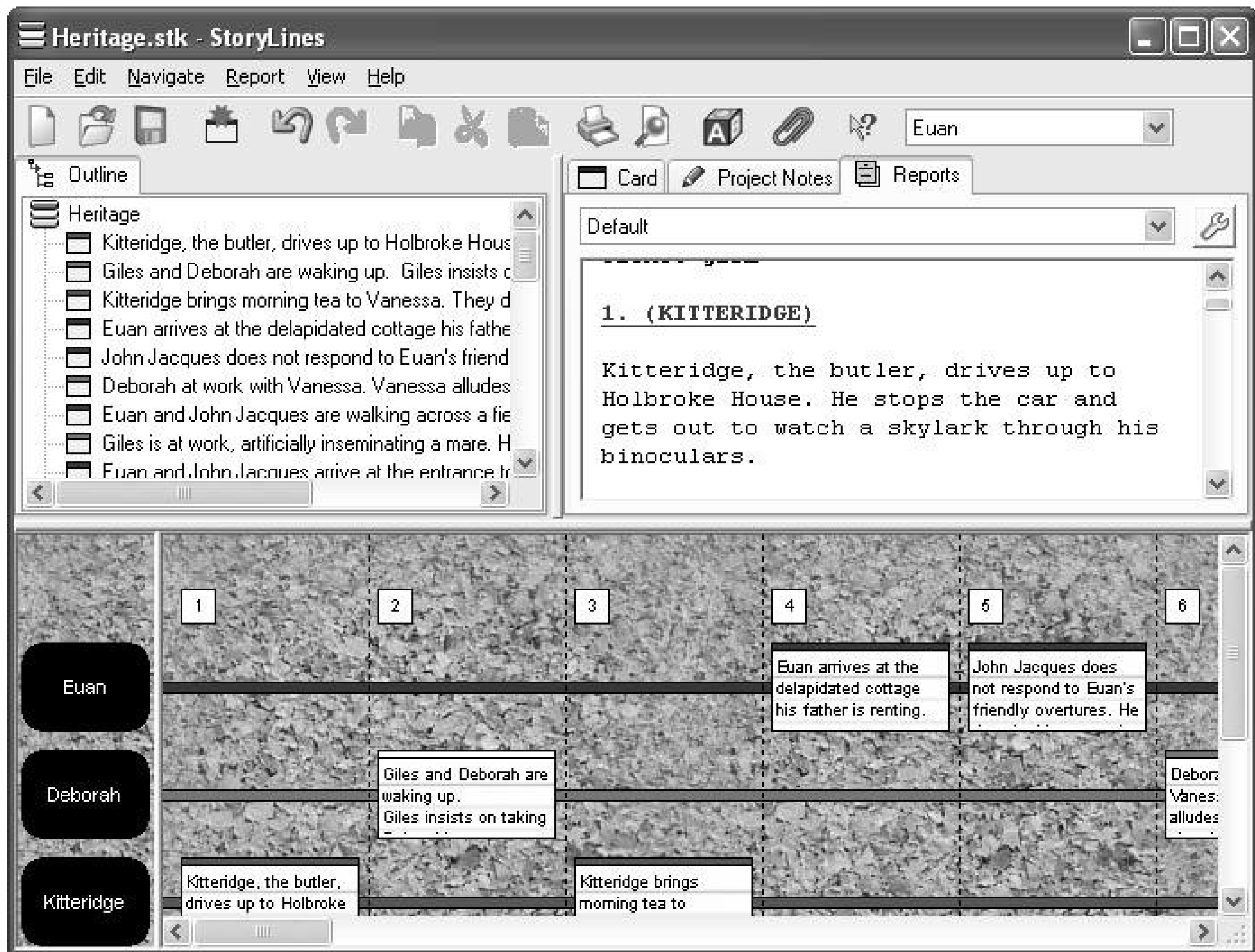
虽然wxWidgets经常被打上图形界面程序开发的标签，但是它在很多其它的应用程序开发方面也提供了很多特性支持。这样作的目的是为了让使用wxWidgets编写的程序的各个部分都可以是跨平台的，而不仅仅是图形界面的部分。这些部分包括：文件和流操作，多线程，程序设置，进程间通讯，在线帮助，数据库访问等等。

1.1 为什么要使用wxWidgets?

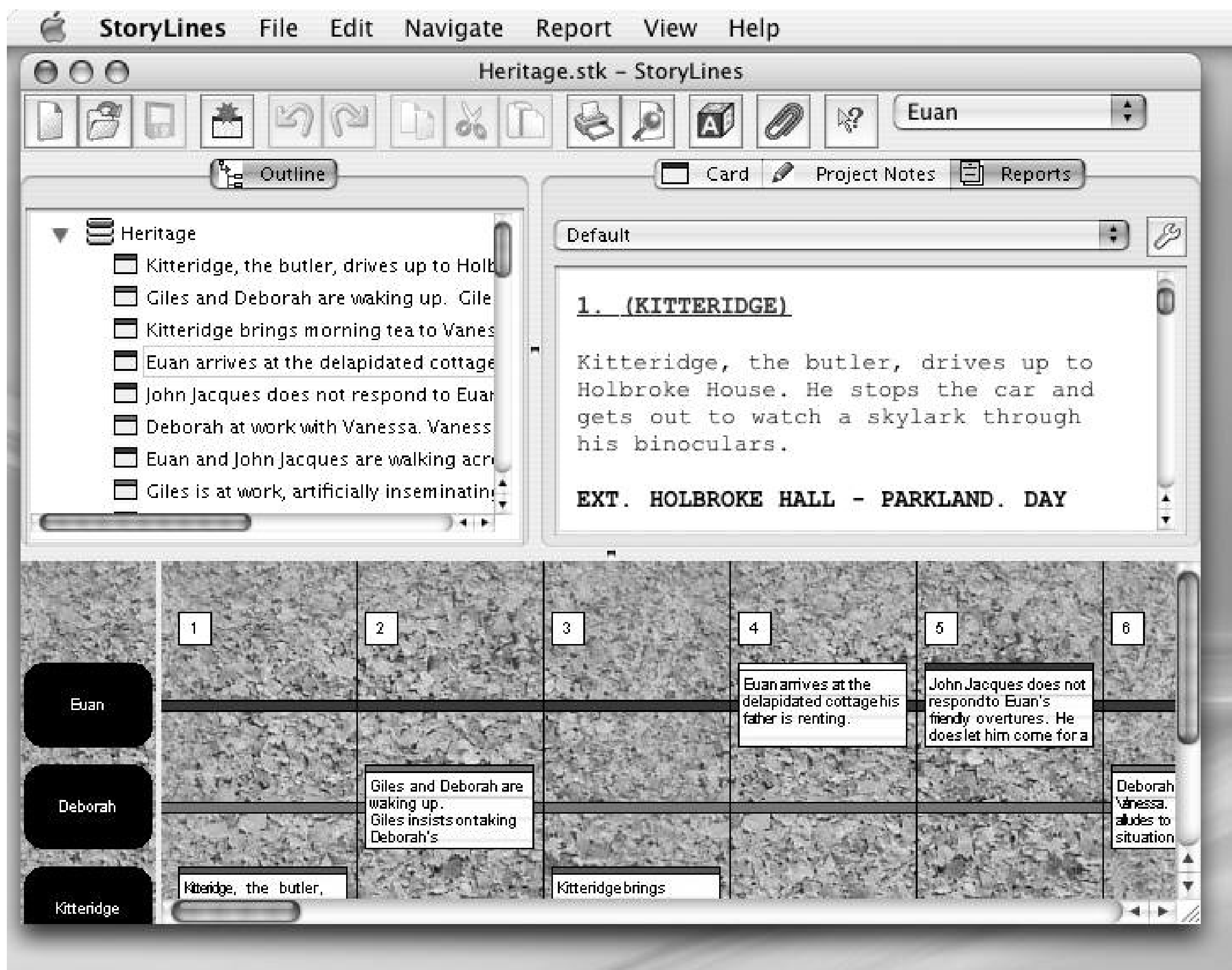
wxWidgets和其它类似的GUI（图形用户界面，下同）库比如MFC或者OWL一个最本质的区别在于，它是跨平台的。wxWidgets提供的 API函数在它支持的所有平台上是想同的或者是非常相似的。这意味着你可以编写一个windows上运行的程序，这个程序不需要经过任何改动，或者只需要 很少的改动（这种情况并不常见），只需要通过重新编译，就可以在Linux或者Max OSX上运行。比起为另外的平台从头编写代码，这显然有很大的好处，另外一个附带的好处是，你不需要重新学习那个平台的API。 而且，你的程序可能在将来很长时间仍然可以使用。因为随着计算机科技的演进，wxWidgets将会随之一起演进，这样你的程序将会很方便的移植到最新的 操作系统以支持最新的特性。

另外一个与众不同的地方在于，wxWidgets可以给你的应用程序提供本地观感。一些其它的可以跨平台的开发框架在不同的平台使用同 样的窗口组件代码（译者注：难到他指的是JAVA？），也许它会通过类似窗口主题这样的方式来模拟本地观感。而wxWidgets则尽可能的使用本地的窗 口控件（当然wxWidgets也提供自己的控件集，这是另外一个话题了），所以wxWidgets的程序不只是看上去象是本操作系统上的原生程序，它实 际上就是原生程序。对于使用应用程序的用户来说，这是非常重要的，因为和本地操作系统标准的任何一点细微的甚至是几乎难以察觉的不同，都会让他们产生避而 远之的想法。

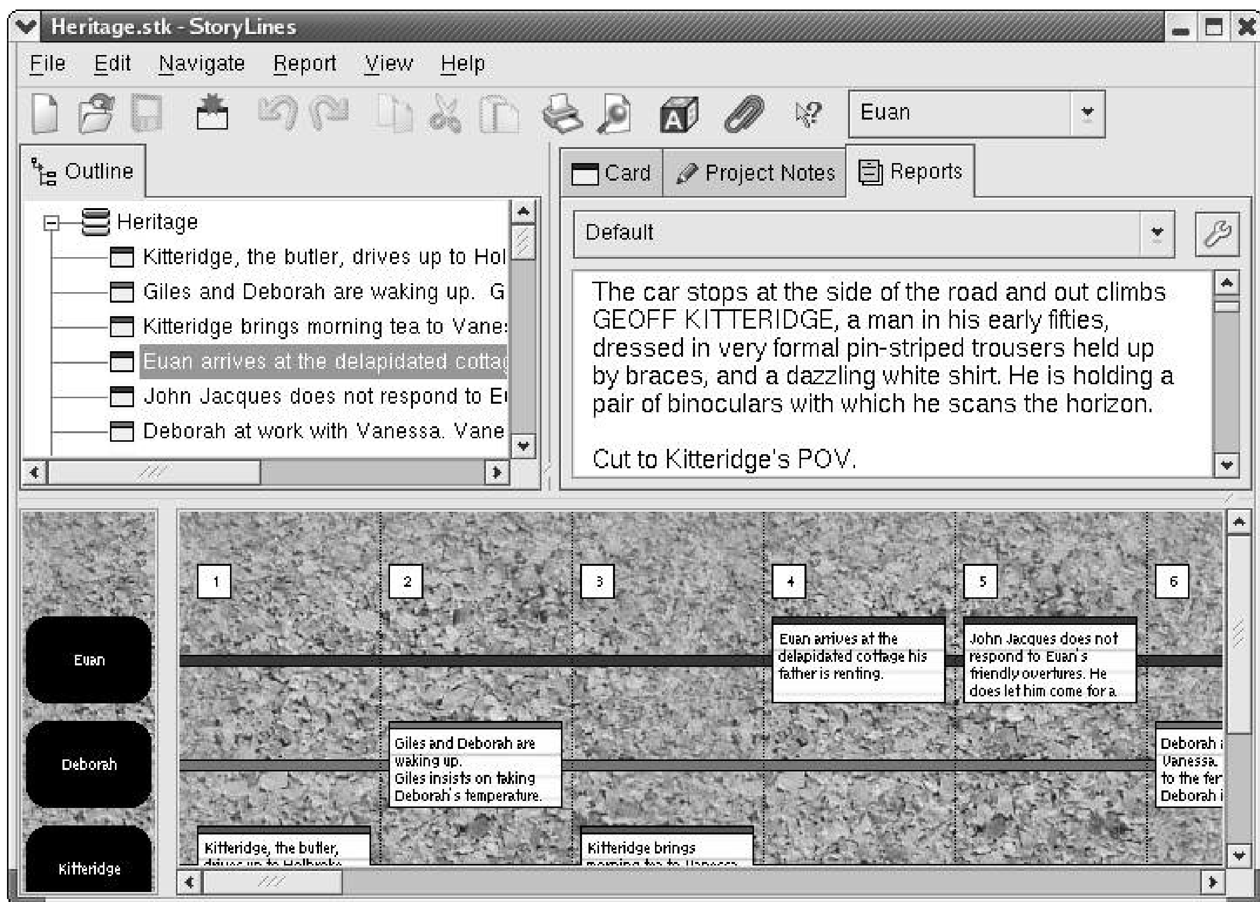
让我们来举例说明。下图演示了一个叫做StoryLines的小程序运行在Windows XP上的样子：



正象大家看到的那样，这是一个典型的Windows应用程序，有典型的Windows的GUI控件例如标签页，滚动条以及下拉列表。类似了，下图演示了这个程序在Max OSX上的样子，正象我们期待的那样，它有着水晶外形图标，没有菜单条（因为按照苹果的风格，当前窗口的菜单条应该显示在屏幕的最顶层。）



最后，我们还将演示一下同样的程序在小红帽Linux上作为一个GTK+程序的样子:



为什么不直接使用JAVA呢？对于基于Web的应用来说，JAVA的确很不错，但是对于桌面应用程序来说，JAVA有时候并不是一个很好的选择。一般来 讲，基于C++的wxWidgets程序会运行更快，感观上更象本地原生程序并且更容易安装，因为它并不依赖于你的机器一定要有JAVA虚拟机。C++也 更容易访问操作系统提供的底层函数并且更容易和已有的C++或者C代码集成。基于以上原因，您现在经常用到的桌面程序中，很少有全部基于JAVA开发的。 而wxWidgets则可以让你开发高性能的，本地原生的应用程序。而这可能正是你的用户所期待的。

wxWidgets是一个开放源代码的项目。毫无疑问，这意味着使用wxWidgets是免费，它不需要额外花费你1分钱（除非您愿意大方的向这个项目进 行捐助），但是，开放源代码并不仅仅意味着免费，它有着更重要的意义。开源项目通常可以持续比它的创建团队或者通常意义上的拥有者更长久的时间。使用 wxWidgets开发程序，你的代码永远不会过时，你的代码所依赖的开发平台永远不会消失。你可以通过直接修改源代码来修正基础库中的问题（译者注：使用Delphi的开发者对此可能有更深的体会，由于众所周知的原因，很多开发工具慢慢的被淘汰了）。你甚至可以自己抽点时间加入到wxWidgets的开 发团队中来，维护其中的一部分代码，这也是一件非常有趣的事情。开源项目的团队成员之所以加入某个团队是因为他们热爱他们正在作的事情，并且迫不及待的想 把他们的知识和别人分享，而商业项目的客服支持人员通常不具有这种理想主义的情节，当你使用wxWidgets开始编程时，你其实是把自己放入一个令人惊 讶的有艺术天赋的一堆天才中间（译者注：我只是按照字面意思翻译，虽然我自己用wxWidgets开发程序，但是这样的话还是让我觉的有一点点善意的恶 心。可能是我的英文太差了，没有理解原话的意思，原话是这样的：When you use wxWidgets, you tap into an

astonishing talent pool, with contributors from a wide range of backgrounds.），这些天才来自世界的各个角落，有着各种各样的背景。开发应用程序需要考虑的很多细节都被这些天才封装在了你可以直接拿来很简 单就可以使用的类中，如果不是这些天才的劳动，你可能要花费很大的精力才能应付。一个开放和活跃的社区将会通过邮件列表对你提供帮助，在这里，你会享受到 讨论的乐趣。这些讨论并不全是和wxWidgets相关的。更多情形下，你是和社区那些有经验的或者没有经验的开发者进行心灵的交流.也许有一天，你会发 现自己成为wxWidgets之所以成功的一分子。

wxWidgets已经被广泛的应用在各种工业领域。它的用户包含了象AOL,AMD,CALTECH, Lockheed Martin, NASA, the Open Source Applications Foundation, Xerox等等这些大的商业和团体机构。wxWidgets拥有很广泛的使用者，从个体的软件开发者到大的商业团体，从计算机科学领域到医疗研究领域，从社会生态学到电信领域。当然，还有数不清的开源项目在使用它，例如Audacity声音编辑项目和pgAdmin III数据库设计和维护项目等。

人们出于各种各样的目的而使用wxWidgets,一些人只是把它作为单平台开发上MFC的优雅的替代者，一些则是为了让他们的程序可以方便的从微软的Windows移植到Linux或者是苹果的OSX。 wxWidgets还正致力于移动终端的支持，包括嵌入式linux，微软的Pocket PC，在不久的将来还会支持Palm OS。

1.2 wxWidgets的历史

1992年，Julian Smart在Edinburgh大学开始制作一个叫做Hardy的图表工具的时候，为了避免其发行版本在Sun的工作站和各种PC之间作选择，他决定使用跨平台的编程框架。但是当时可选的跨平台的编程框架不多，而他的部门也不可能给他很多的预算，所以他只能自己创建一个自己的跨平台编程框架。这样， wxWidgets 1.0诞生了。 1992年9月，学校允许他把他的wxWidgets 1.0上传到部门的FTP服务器，因此别的一些开发者也开始使用他的代码。最开始的时候，wxWidgets是面向XView和MFC 1.0的，由于Borland C++的适用者抱怨其对MFC的依赖，所以Julian Smart用纯Win32的代码重写了wxWidgets。又因为XView很快被Motif取代，很快，Widgets提供了对Motif的支持。

不久以后，一个很小但是却很付有激情的wxWidgets用户社区成立了并且拥有了自己的邮件列表。大量的新代码和补丁开始融入到wxWidgets中，其中包括Markus Holzem提供的Xt的支持。wxWidgets也自然的拥有了越来越多的来自世界各地的使用者：独立工作者，学术机构，政府机构以及很多企业用户等，他们认为wxWidgets提供的产品质量和产品支持甚至好过他们见过的或者用过的其它商业的产品。

1997年，在Markus Holzem的帮助下，新版的wxWidgets 2 API问世。此时，Wolfram Gloger建议应该提供GTK+的支持。GTK+是被GNOME桌面系统采纳的一套窗口控件。于是，Robert Roebling开始领导GTK版本的wxWidgets的开发，现在wxWidgets的GTK版本已经成为其在UNIX/LINUX下的最主要的版本。到了1998年，Windows和GTK+的版本被合入版本控制工具CVS。Vadim Zeitlin加入到项目中来帮助管理和维护如此大量的设计和代码，同年，Stefan Csomor开始着手增加对Mac OS的支持。

1999年，Vaclav Slavik的令人印象深刻的wxHTML类和HTML帮助文件显示控件被加入进来。2000年，SciTech公司开始开发wxUniversal版本，这个版本提供属于wxWidgets自己的不依赖于任何其它图形库的窗口控件，以便支持那些没有原生窗口控件库的操作系统。wxUniversal最初被用于SciTech公司的MGL产品，这个产品为图形用户界面提供了底层支持。

到了2002年，Julian Smart和Robert Roebling在wxUniversal的基础上提供了wxX11版本，这个版本仅依赖于Unix和X11,因此它几乎适用于任何的类Unix环境，所以，它可以被用在相当底层的系统中。

2003年，wxWidgets开始了对Windows CE的支持，同年Robert Roebling在GPE嵌入式Linux平台上演示了使用wxGTK编写的程序。

2004年，因为收到微软的商标方面的威胁，wxWidgets被迫从它原来的名字"wxWindows"改名。

同样是在2004年，Stefan Csomor和一大群热心的参与者彻底的修改了wxMac OSX版本，OSX版本的功能和性能都得到了极大的提升。而David Elliot领导的小组正在稳步的开发一个基于Cocoa的版本，William Osborne也着手开发一个可以支持wxWidgets的"minimal"例子的Palm OS 6的版本。 2005年4月，2.6版的wxWidgets发布了，几乎所有的平台版本在这个版本都有了大幅的改进和提高。

wxWidgets将来的计划包括：

- 一个包管理工具，使得集成第三方工具变得容易。
- 更好的嵌入式支持。
- 更好的事件处理机制。
- 增强型控件支持：比如一种捆绑了树形控件和列表控件的控件。
- wxHTML 2提供在各种平台下的完整的Web能力支持。
- STL标准兼容
- 完整的Palm OS支持

1.3 wxWidgets社区

wxWidgets社区是非常活跃的。它拥有两个邮件列表: wx-users (用于普通用户)和 wx-dev (用于wxWidgets的开发者).一个网站, 网站上有最新消息,一些文章以及和wxWidgets有关的链接, 还拥有一个"Wiki," 所谓"Wiki"是一个网页的集合, 这些网页可以被任何人修改和增加信息.还有一个论坛可以用来就某一话题发起讨论. 这些网络资源的网址列举在下面：

- http://www.wxwidgets.org: 这是wxWidgets的官方网站
- http://lists.wxwidgets.org: wxWidgets的邮件列表
- http://wiki.wxwidgets.org: wxWidgets的Wiki
- http://www.wxforum.org: wxWidgets的论坛

和大多数开放源代码的项目一样, wxWidgets采用CVS来进行代码的管理和修改记录的跟踪. 为了保证开发的有序进行, 只有少数几个开发者拥有对 CVS库的修改权限, 其它的开发者可以通过提交补丁和提交缺陷报告的方式参与开发, wxWidgets目前使用的补丁和缺陷管理系统是由 SourceForge提供的。CVS库主要有两个分枝, 稳定版分支和开发分支。CVS的稳定版分支只允许为了修改缺陷而进行修改。新功能的开发需要在开发分支进行。稳定版的版本号都是双数的, 例如2.4.x等, 而开发分支的版本好都是单数的, 比如2.5.x等。对于单数版本, 使用者最好等待新的稳定版本的发布再使用。当然直接从CVS下载最新的开发版本也是可以的。

wxWidgets的API的修改通常是由开发者在wx-dev邮件列表里讨论以后决定的。

除了以上这些, 很多其它的wxWidget相关的项目都拥有他们自己的社区。例如wxPython社区和wxPerl（参见附录E：wxWidgets的第三方工具）社区.

1.4 wxWidgets和面向对象编程

和大多数现代的GUI编程框架一样，wxWidgets大量使用了面向对象编程的概念。每一个窗口都是一个C++的对象。这些对象已经被预置了很好的处理机制，可以接收事件并对事件作出相应的反应。用户所看到的，就是这个对象的交互系统中可视化那一部分。作为一个程序开发人员，你所要作的事情就是合理的安排这些可视的行为集来让它们作出的反应看上去更合理。wxWidgets已经实现了很多默认的行为来让这个工作变的更容易。

当然，面向对象的思想和GUI编程并不是同时产生的。但是在20世纪70年代，由Alan Kay和其它一些人设计的面向对象的语言SmallTalk却是GUI编程历史上的一个重要的里程碑。无论对于用户界面设计来说，还是对计算机程序语言来说，它都是一个创新。虽然wxWidgets使用不同的语言和不同的API，但是就面向对象的原理来说，本质上都是一样的。

1.5 wxWidgets的体系结构

下表展示的wxWidgets的四层体系结构: wxWidgets公用API层，各个平台发行版，用于各个平台的API和操作系统层。

wxWidgets API						
wxWidgets Port	wxMSW	wxGTK	wxX11	wxMotif	wxMac	
Platform API	Win32	GTK+	Xlib	Motif/Lesstif	Carbon	
Operating System	Windows/Windows CE	Unix/Linux			Mac OS 9/ Mac OS X	

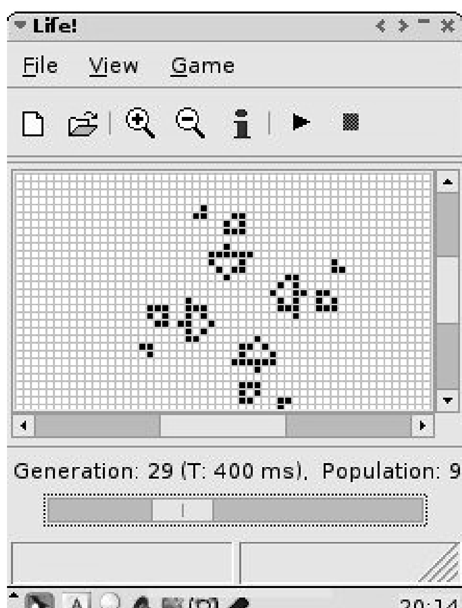
下面依次说明目前已有的各个平台发行版本：

wxMSW

这个版本编译和运行在各个版本的微软的Windows操作系统上，包括：Windows95,Windows98,WinMe,Windows NT,Windows 2000,Windows Xp以及Windows 2003.在linux平台上，这个版本也可以使用Wine的库进行编译，并且可以被配置成在WinCE上运行。除了使用本地原生窗口控件，这个版本也可以配置成使用wxWidgets自己的窗口控件。

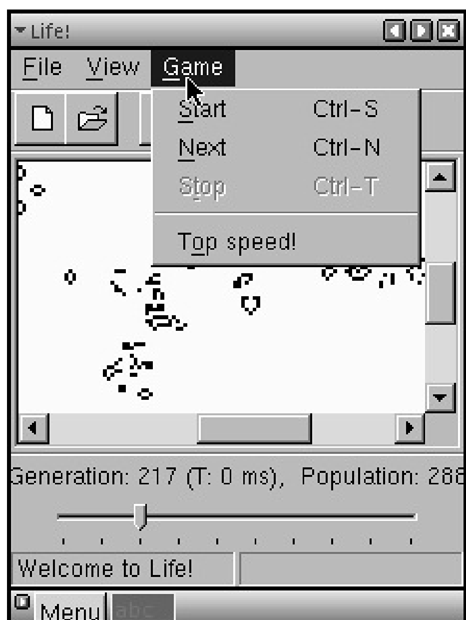
wxGTK

wxWidgets的GTK+版本可以使用GTK的1.x或者2.x版本,支持所有可以运行X11和GTK的类Unix平台 (比如： Linux, Solaris, HP-UX, IRIX, FreeBSD, OpenBSD, AIX等). 它也可以运行在那些有足够资源的嵌入式平台，比如GPE Palmtop环境 (如下图所示). wxWidgets的GTK版本是类Unix系统的推荐版本.



wxX11

wxWidgets的X11版本使用了wxUniversal的窗口控件集，直接运行在Xlib上。这使得它很适合嵌入式系统,当然它也可以运行在那些不喜欢GTK+的桌面系统上。它支持所有可以运行X11的Unix系统，当然wxX11并不像wxGTK那样完善。下图演示了Life程序使用wxX11版本编译运行在一个iPAQ PDA的类似Linux/TinyX环境下的样子。



wxMotif

这个版本的wxWidgets可以在大多数拥有 Motif, OpenMotif, 或者Lesstif的Unix系统上. 既然连Sun自己都正准备把它的窗口控件集转向GNOME和GTK+,对于大多数开发者来说，Motif并不是一个很可靠的选择.

wxMac

wxMac是为Mac OS 9 (9.1以后的版本)和Mac OS X (10.2.8以后的版本)准备的. 如果在Mac OS 9上编译,你需要 Metrowerks CodeWarrior的工具包, 如果在Mac OS X上编译, 你可以选择Metrowerks CodeWarrior工具包或者苹果公司的工具包. 如果使用苹果公司的工具包，你应该使用Xcode 1.5或者更高的版本,或者你可以考虑直接使用命令行工具toolsGCC 3.3或者其后续的版本.

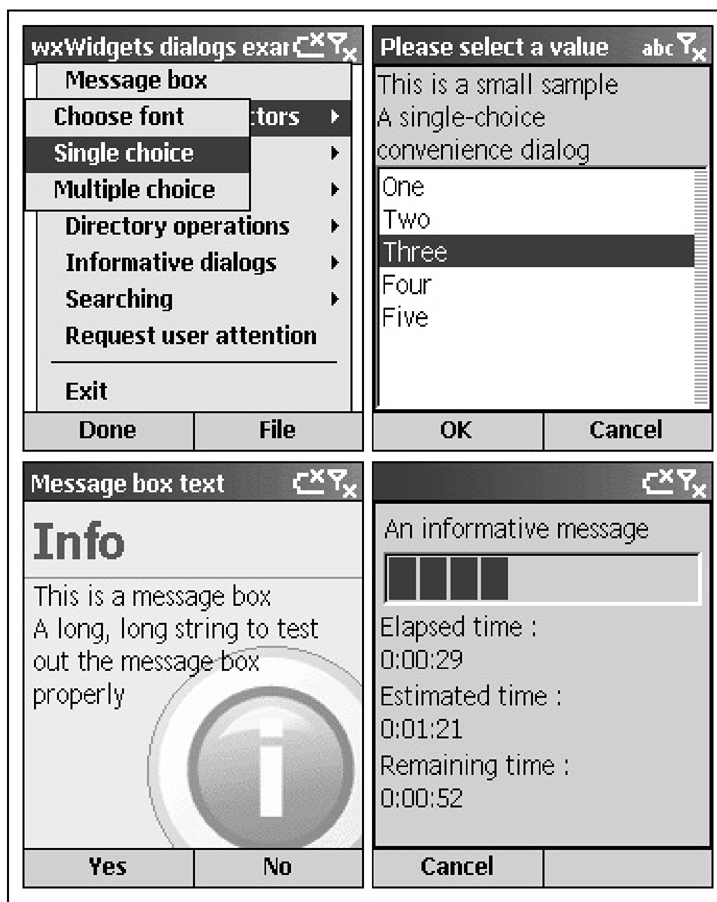
wxCocoa

这是一个正在进行中的版本, 它使用Mac OS X的Cocoa API. 虽然Carbon和Cocoa的功能很相似,但是这个版本有可能会支持除Mac以外的其它支持GNUStep的操作系统。

wxWinCE

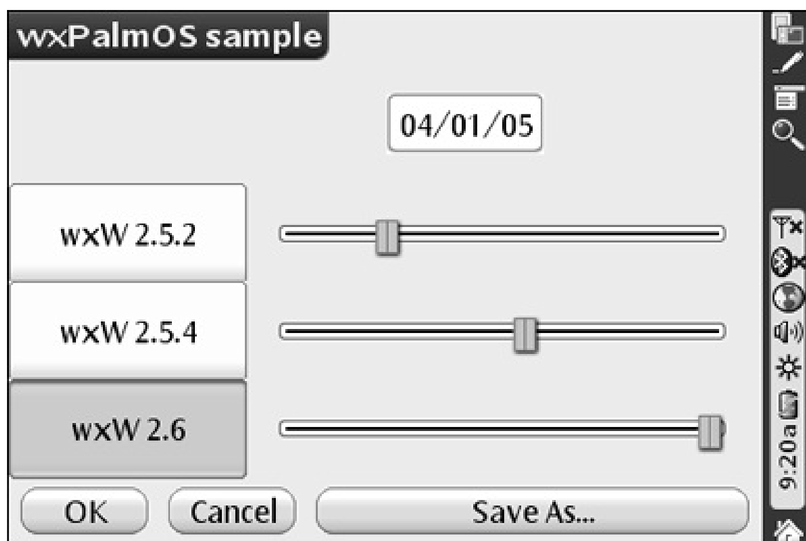
Windows CE版本的wxWidgets封装了WindowsCE平台上的各种不同的开发包,包括Pocket PC和Smartphone等. 这个版本包含在wxMSW的Win32版本中。下面第一副图演示了Life程序在Pocket PC 2003模拟器上运行的样子。第二副图则演示了wxWidgets中的对话框例子运行在一个拥有四个屏幕，分辨率为176x220的 SmartPhone2003上的样子。wxWidgets作了大量的用户界面适配方面的工作，比如因为Smartphone只支持两个菜单按钮，所以在通常显示菜单的地方wxWidgets构建了可以折叠的菜单。尽管如此，一些地方仍然需要依靠编程者使用不同的代码，比如应该使用SetLeftMenu 和SetRightMenu函数来代替直接在对话框上增加两个确定和取消按钮。





wxPalmOS

这个版本是为Palm OS 6准备的。到作者写这本书的时候为止，这个版本还处在很初级的阶段，但是已经可以在Palm OS 6的模拟器中运行一个很简单的小程序了。参见下图：



wxOS2

wxOS2是一个由别人维护的用于OS/2或者eComStation的版本(is a Presentation Manager port for OS/2 or eComStation).

wxMGL

这个版本使用了SciTech公司的底层图形库,窗口控件使用的是wxUniversal中的版本。

内部组织

在内部，wxWidgets的代码大致分为6层：

1. 通用代码被所有的版本使用，包括类的数据结构，运行期类型信息，和一些公共基类比如wxWindowBase等，这些基类的代码将被所有它的子类所继承。
2. 一般代码用来实现独立于各个平台的高级窗口控件，在某个平台不具有某种控件的时候将使用这部分代码，比如wxWizard和wxCalendarCtrl。
3. 通用组件基本的窗口控件集，这套控件可以在某个平台（比如X11或者MGL）不具有它自己的窗口控件的时候使用。
4. 平台相关代码调用特定平台的API来实现某个类的代码。比如在wxMSW中的wxTextCtrl控件的实现是封装了Win32的edit控件。
5. 外来代码存放在一个单独的contrib目录中，提供一些非必要但是很有用的类实现。比如wxStyledTextCtrl
6. 第三方代码不是由wxWidgets开发维护但是被wxWidgets使用以提供一些很重要的特性的代码，比如JPEG, Zlib, PNG和Expat库。

每一个平台版本所需要作的事情就是提取它需要的那些层的代码，然后用它那个平台的底层的API来实现wxWidgets的API。

当你编译你的代码的时候，wxWidgets怎么知道要编译哪一个平台的类呢？当你包含一个wxWidgets的头文件（比如 wx/textctrl.h）的时候,由于使用的不同的宏定义，实际上你包含的是一个特定平台的头文件(wx/msw/textctrl.h)。然后，当你链接wxWidgets的库文件的时候，当然这个库文件也需要是用同样的宏定义编译的。你可以同时拥有多套宏定义，例如你可以有调试版本和发布版本两套宏定义。通过这些宏定义的不同，你可以控制编译器链接wxWidgets的动态或者是静态版本，也可以禁止编译某个特定组件，或者是决定你要编译的是Unicode版本还是ANSI版本。为了达到这个目的，你需要修改setup.h或者增加不同的编译选项，这取决于你的编译器。有关这些问题更详细的描述，请参见附录A，???安装wxWidgets???

另外一点提示是：虽然wxWidgets封装了本地的平台相关的API，并不意味着在你的wxWidgets程序中不可以使用这些API。只是，在绝大多数情况下，你用不着使用这些API。

1.6 许可协议

wxWidgets采用的是L-GPL的许可协议外加一个附加条款。你可以从wxWidgets的网站上或者wxWidgets的分发包的docs目录里看到这份许可协议。这份协议总体上来说就是：你既可以使用wxWidgets开发自由软件，也可以使用它开发商业软件，并且不需要支付任何版权费用。你既可以动态链接wxWidgets的运行期库文件，也可以使用静态链接。如果你对wxWidgets本身进行了任何改动，你必须公开这一部分的源代码。而完全属于你自己的那部分代码或者库文件则不需要公开。另外，在发布使用wxWidgets编写的软件的时候，你还需要考虑wxWidgets的一些可选组件自己的许可协议，比如PNG和JPEG图形库的许可协议，它们和wxWidgets的许可协议可能并不完全相同。

本书中所有的例子和源代码同样使用wxWidgets的许可协议。

第一章小结

在这一章里，我们试图向你解释wxWidgets是什么，简单的描述了它的历史，大概支持哪些平台以及大概说了一下它的内部的组织结构。

在下一章 "开始使用"里，我们会通过几个例子来展示一下用wxWidgets编写应用程序大概是怎样一个样子。

第二章 开始使用

在这一章里，我们将使用一个很小的例子，来建立你对用wxWidgets编程的一个大致的印象。我们将会来看看一个用wxWidgets编写的程序是怎么开始的又是怎样结束的，怎样创建主窗口，怎样响应用户的命令。正如wxWidgets所追求的短小精悍的哲学一样，这就是所有这一章我们要涉及的内容。不过在这之前，你可能先要参考附录A来安装wxWidgets.

2.1 一个小例子

下图演示了我们的小例子在Windows上运行的样子



这个小例子创建了一个主窗口(是一个wxFrame类的实例),这个主窗口有一个菜单条和一个状态条。菜单条上的菜单按照你的命令显示一个关于窗口或者退出这个小程序。显然这算不上什么杀手级的大程序，但是足以给你展示wxWodgets的一些基本原则,并且让你可以相信，随着知识的慢慢积累，有一天你将作出更复杂的（那些杀手级的）程序，

2.2 应用程序类

每一个wxWidgets程序都需要定义一个wxApp类的子类，并且需要并且只能构造一个这个类的实例，这个实例控制着整个程序的执行。你的这个继承自 wxApp的子类至少需要定义一个OnInit函数，当wxWidgets准备好运行你写的代码的时候，它将会调用这个函数（和一个典型的Win32程序中的main函数或者WinMain函数类似）。

你定义这个子类的代码可能和下面的代码类似：

```
class MyApp : public wxApp
{
  public:
      virtual bool OnInit();
};
```

在这个OnInit函数中，你通常应该创建至少一个窗口，对传入的命令行参数进行解析，为应用程序进行数据设置和其它的一些初始化的操作.如果这个函数返回真，wxWidgets将开始事件循环用来处理用户输入并且在必要的情况下处理这些输入。如果OnInit函数返回假，wxWidgets将会释放它内部已经分配的资源，然后结束整个程序的运行。

接下来我们看一个最简单的OnInit函数的实现：

```
bool MyApp::OnInit()
{
   MyFrame *frame = new MyFrame(wxT("Minimal wxWidgets App"));
   frame->Show(true);
   return true;
}
```

你可能还会注意到上面例子中的wxT这个宏，在接下来的例子中，这个宏还会被频繁用到。它的作用是让你的代码兼容Unicode模式。这个宏和另外一个*T*宏的作用是完全一样的。使用这个宏也不会带来运行期的性能损失。（你可能还会遇到另外一个类似的"()"标记，这个标记是用来告诉 wxWidgets将其中的字符串翻译成其它语言的版本，参见第16章“编写国际化程序”）。

那么创建MyApp的实例的代码在哪里呢？实际上，这是在wxWidgets内部实现的，不过你仍然需要告诉wxWidgets需要创建哪一个App类的实例，所以你还需要增加下面的一个宏:

```
IMPLEMENT_APP(MyApp)
```

如果没有实现这个类，wxWidgets就不知道怎样创建一个新的应用程序对象。这个宏除了上述的功能以外，还会检查编译应用程序使用的库文件是否和当前的库文件的版本相匹配，如果没有这种检查，由此而产生的一些运行期的错误可能很难被查出原因。

当wxWidgets创建这个MyApp类的实例的时候，会将创建的结果赋值给一个全局变量wxTheApp.你当然可以在你的程序中使用这个变量，但是你可能不得不一遍又一遍的进行从wxApp到MyApp的类型强制转换。增加下面的这一行声明以后，你就可以调用wxGetApp()函数，这个函数会返回一个到这个MyApp实例的引用，这样用起来就方便多了。

```
DECLARE_APP(MyApp)
```

一点提示：

即使没有声明DECLARE_APP,你仍然可以不用进行类型强制转化就直接对wxTheApp变量调用wxApp的方法.这可以避免在所有的头文件中包含MyApp的头文件，对于那些库文件而不是应用程序的代码来说也更有意义，而且还可以缩短编译的时间。

2.3 Frame窗口类

我们来看一看自定义的Frame窗口类MyFrame.一个Frame窗口是一个可以容纳别的窗口的顶级窗口，通常拥有一个标题栏和一个菜单栏。下面是我们的例子中这个类的定义，可以将其放在MyApp的定义之后：

```
class MyFrame : public wxFrame
{
public:
    MyFrame(const wxString& title);
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);
private:
    DECLARE_EVENT_TABLE()
};
```

这个窗口类的定义有一个构造函数，两个用来把菜单命令和C++代码相连的事件处理函数，还有一个宏来告诉wxWidgets这个类想要自己处理某些事件。

2.4 事件处理函数

你也许已经注意到了，事件处理函数在MyFrame类中不是虚函数。如果不是虚函数，他们是怎样被调用的呢？答案就在下面的事件表里：

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(wxID_ABOUT, MyFrame::OnAbout)
    EVT_MENU(wxID_EXIT,  MyFrame::OnQuit)
END_EVENT_TABLE()
```

所谓事件表，是一组位于类的实现文件（.cpp文件）中的宏，用来告诉wxWidgets来自用户或者其它地方的事件应该怎样和类的成员函数对应起来。

前面展示的事件表表明，要把鼠标点击标识分别为wxID_EXIT和wxID_ABOUT的菜单项的事件和MyFrame的成员函数OnAbout和 OnQuit关联起来。这里的EVT_MENU宏只是很多中事件宏的其中之一，事件宏的作用是告诉wxWidgets哪种事件应该被关联到哪个成员函数。这里的两个标识wxID_ABOUT和wxID_EXIT是wxWidgets预定义的宏，通常你应该通过枚举，常量或者预编译宏的方式定义你自己的标识。

用上面的方法定义的时间表是一种静态的事件表，它不可以在运行期改变，在下一章里，我们将描述怎样定义可以在运行期改变的动态事件表。

现在，让我们来看看这两个事件处理函数.

```
void MyFrame::OnAbout(wxCommandEvent& event)
{
    wxString msg;
    msg.Printf(wxT("Hello and welcome to %s"),
               wxVERSION_STRING);
    wxMessageBox(msg, wxT("About Minimal"),
                 wxOK | wxICON_INFORMATION, this);
}


void MyFrame::OnQuit(wxCommandEvent& event)
{
    Close();
}
```

当用户点击关于菜单项的时候，MyFrame::OnAbout函数弹出一个消息框。这用到了wxWidgets提供的API wxMessageBox,它的四个参数分别代表消息内容，标题，窗口类型以及父窗口。

当用户点击退出菜单项的时候，MyFrame::OnQuit函数被调用（你已经意识到了，这是事件表的功劳）。它调用wxFrame类的Close函数来释放frame窗口。因为没有别的窗口存在了，这触发了应用程序的退出，实际上，wxFrame类的Close函数并不直接关闭 frame窗口，而是

产生一个wxEVT_CLOSE_WINDOW事件，这个事件默认的处理函数调用wxWindow::Destroy函数释放了 frame窗口。

用户还可以通过别的方法关掉应用程序，比如通过点击标题栏上的关闭按钮或者是通过系统菜单中的关闭菜单，在这种情况下，OnQuit函数是怎样被调用的呢？事实上，在这种情况，OnQuit函数并没有被调用。这时，wxWidgets会通过Close函数（象OnQuit中的那样），给frame窗口发送一个wxEVT_CLOSE_WINDOW事件，这个事件默认的处理函数会释放掉frame窗口。在你的应用程序中，可以通过重载这个处理函数来增加改变这种默认的行为，比如：如果你想问一问你的用户是不是真的要关闭窗口。关于这种处理的细节，可以参见第4章,"窗口基础"。

另外，大多数的应用程序类还应该重载一个OnExit函数，以便在任何时候程序退出时，执行一下清理和资源回收的动作。需要注意的是，这个函数只有在OnInit函数返回真的时候才会被执行。当然，我们这个小例子程序就用不着定义这个函数了。

2.5 Frame窗口的构造函数

最后，让我们来看看Frame窗口的构造函数，正是它实现了frame窗口的图标，菜单条和状态条。

```
#include "mondrian.xpm"


MyFrame::MyFrame(const wxString& title)
       : wxFrame(NULL, wxID_ANY, title)
{
    SetIcon(wxIcon(mondrian_xpm));
    wxMenu *fileMenu = new wxMenu;
    wxMenu *helpMenu = new wxMenu;
    helpMenu->Append(wxID_ABOUT, wxT("&About...\tF1"),
                     wxT("Show about dialog"));
    fileMenu->Append(wxID_EXIT, wxT("E&xit\tAlt-X"),
                     wxT("Quit this program"));
    wxMenuBar *menuBar = new wxMenuBar();
    menuBar->Append(fileMenu, wxT("&File"));
    menuBar->Append(helpMenu, wxT("&Help"));
    SetMenuBar(menuBar);
    CreateStatusBar(2);
    SetStatusText(wxT("Welcome to wxWidgets!"));
}
```

这个构造函数首先调用它的基类（wxFrame）的构造函数，使用的参数是父窗口(还没有父窗口，所以用NULL),窗口标识(wxID_ANY标识让 wxWidgets自己选择一个)和标题。这个基类的构造函数才真正创建了一个窗口的实例。除了这样的调用方法，还有另外一种方法是直接在构造函数里面显式调用基类默认的构造函数，然后调用wxFrame::Create函数来创建一个frame窗口的实例。

小图片或者是图标在所有的平台上都可以用XPM格式来表示。XPM文件其实是一个ASCII编码的完全符合C++语法的文本文件，所以可以直接用C++的方式包含到代码中（译者注：显然这样的包含方式在分发软件的时候是不需要分发这个图片文件的）。SetIcon那一行代码使用mondrian_xpm变量在堆栈上创建了一个图标（这个mondrian变量是在mondrian.xpm文件里定义的）。然后将这个图标和 frame窗口关联在一起。

接下来创建了菜单条。增加菜单项的Append函数的三个参数的意义分别为：菜单项标识，菜单上的文本以及一个稍微长一些的帮助字符串。这个帮助字符串会自动在菜单项被高亮显示的时候自动显示在状态栏上。菜单上的文本中由"&"符号前导的字符将成为菜单的快捷操作符，在实际的显示中用下划线表示。而"\t"符号则前导一个全局的快捷键，这个快捷键甚至可以在菜单项没有显示的时候触发菜单功能。

这个构造函数所做的最后一件事是创建一个由两个区域组成的状态条并且在状态条的第一个区域写上欢迎的字样。

2.6 完整的例子

现在是时候把所有的代码放在一起了，通常，我们应该把头文件和实现文件分开，但是对于这样小的一个程序，就没有这个必要了。

```
// Name:        minimal.cpp
// Purpose:     Minimal wxWidgets sample
// Author:      Julian Smart


#include "wx/wx.h"

// 定义应用程序类
class MyApp : public wxApp
{
public:
    // 这个函数将会在程序启动的时候被调用
    virtual bool OnInit();
};

// 定义主窗口类
class MyFrame : public wxFrame
{
public:
    // 主窗口类的构造函数
    MyFrame(const wxString& title);

    // 事件处理函数
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);

private:
    // 声明事件表
    DECLARE_EVENT_TABLE()
};

// 有了这一行就可以使用 MyApp& wxGetApp()了
DECLARE_APP(MyApp)

// 告诉wxWidgets主应用程序是哪个类
IMPLEMENT_APP(MyApp)

// 初始化程序
bool MyApp::OnInit()
{
    // 创建主窗口
    MyFrame *frame = new MyFrame(wxT("Minimal wxWidgets App"));

    // 显示主窗口
    frame->Show(true);

    // 开始事件处理循环
    return true;
}

// MyFrame类的事件表
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(wxID_ABOUT, MyFrame::OnAbout)
    EVT_MENU(wxID_EXIT,  MyFrame::OnQuit)
END_EVENT_TABLE()

void MyFrame::OnAbout(wxCommandEvent& event)
{
    wxString msg;
    msg.Printf(wxT("Hello and welcome to %s"),
```

```
                 wxVERSION_STRING);

    wxMessageBox(msg, wxT("About Minimal"),
                   wxOK | wxICON_INFORMATION, this);
}

void MyFrame::OnQuit(wxCommandEvent& event)
{
    // 释放主窗口
    Close();
}

#include "mondrian.xpm"

MyFrame::MyFrame(const wxString& title)
       : wxFrame(NULL, wxID_ANY, title)
{
    // 设置窗口图标
    SetIcon(wxIcon(mondrian_xpm));

    // 创建菜单条
    wxMenu *fileMenu = new wxMenu;

    // 添加“关于”菜单项
    wxMenu *helpMenu = new wxMenu;
    helpMenu->Append(wxID_ABOUT, wxT("&About...\tF1"),
                     wxT("Show about dialog"));

    fileMenu->Append(wxID_EXIT, wxT("E&xit\tAlt-X"),
                     wxT("Quit this program"));

    // 将菜单项添加到菜单条中
    wxMenuBar *menuBar = new wxMenuBar();
    menuBar->Append(fileMenu, wxT("&File"));
    menuBar->Append(helpMenu, wxT("&Help"));

    // ...然后将菜单条放置在主窗口上
    SetMenuBar(menuBar);

    // 创建一个状态条来让一切更有趣些。
    CreateStatusBar(2);
    SetStatusText(wxT("Welcome to wxWidgets!"));
}
```

2.7 wxWidgets程序一般执行过程

下面大概的描述一下整个程序的执行过程：

1. 依照系统平台的不同，不同的main函数或者winmain函数或者其它类似的函数被调用(这个函数是由wxWidgets提供的,而不是由应用程序提供的).wxWidgets 初始化它自己的数据结构并且创建一个MyApp的实例.
2. wxWidgets调用MyApp::OnInit函数, 这个函数会创建一个MyFrame的实例.
3. MyFrame的构造函数通过它的基类wxFrame的构造函数创建一个窗口，然后给这个窗口增加图标，菜单栏和状态栏.
4. MyApp::OnInit函数显示主窗口并且返回真.
5. wxWidgets开始事件循环，等待事件发生并且将事件分发给相应的处理过程.

就目前我们所知道的，应用程序会在以下情况下退出：主窗口被关闭，用户选择退出菜单或者系统按钮和系统菜单中的关闭选项（这些系统菜单和系统按钮在不同的系统中就往往千差万别了）。

2.8 编译和运行程序

你可以在附带光盘的examples/chap02里找到这个例子。你可能需要把它拷贝到你硬盘上的某个目录中才能进行编译。因为要提供适合所有读者的软件环境的Makefile几乎是不可能的，我们提供了一个DialogBlocks的项目文件，并且把我们能想到的大多数系统平台进行了配置。你可以参考附录C，“使用DialogBlocks创建应用程序”来找到适合你自己的编译器的编译方法。在附录B中，我们也大概的描述了直接用wxWidgets编译的方法。

你可以从CDRom上安装wxWidgets和DialogBlocks，如果你使用的是windows操作系统并且你的系统中还没有可以用的编译器，你还需要安装一个编译器。然后运行DialogBlocks，并且在其设置叶面设置你的wxWidgets和编译器的路径，然后打开 examples/chap02/minimal.pj，选择适合你的编译器和编译平台，比如Mingw Debug或者VC++ Debug(Windows),GCC Debug GTK+(Linux),或者GCC DEBUG Max(Max Os X)等，然后点击绿色的编译和运行工程按钮。如果你的wxWidgets库还没有被编译，它将首先编译wxWidgets库。

你也可以在wxWidgets的sample/minimal目录中找到一个类似的例子。如果你不想使用DialogBlocks编译，你也可以试着编译这个例子。你可以在附录A“安装wxWidgets”中找到编译这个例子的方法，

第二章小结

本章向你展示了一个非常简单的wxWidgets小程序是怎样工作的。我们已经接触到了wxFrame类，事件处理，应用程序初始化，并且创建了一个菜单条和状态条。虽然代码看上去有些复杂，但是任何其它更复杂的程序设计和我们展示在这个小例子中的一些公共的原则都是相同的。在下一章里，我们会更近距离的看一看事件表机制以及我们的应用程序是怎样处理事件表的。

第三章 事件处理

3.1 事件驱动编程

当程序员们首次面对苹果公司的第一个图形界面个人电脑的时候，他们为这种和以前所有的经验都不同的电脑操作方法感到惊奇。鼠标指针在一个个的窗口之间移来移去，滚动条，菜单，文本编辑框等等等等，真的很难以想象，这么多让人眼花缭乱的东西，其背后的代码该是多么复杂和不可思议。似乎所有这一切都是以完全并行的方式运行的，虽然这只是一个假象。对于很多人来说，苹果个人电脑是他们对事件驱动编程的第一印象。

所有的GUI程序都是事件驱动的。换句话说，应用程序一直停留在一个循环中，等待着来自用户或者其他地方（比如窗口刷新或网络连接）的事件，一旦收到某种事件，应用程序就将其扔给处理这个事件的函数。虽然看上去不同的窗口是同时被刷新的，但实际上，绝大多数的GUI程序都是单线程的，因此窗口的刷新是依次按顺序进行的。如果由于某种意外你的电脑变得很慢导致窗口刷新的过程变的很明显，你就会注意到这一点。

不同的GUI编程架构用不同的方法将它内部的事件处理机制展现给程序开发者。对于wxWidgets来说，事件表机制是最主要的方法。在下一小节我们会对此进行进一步的解释。

3.2 事件表和事件处理过程

wxWidgets事件处理系统比起通常的虚方法机制来说要稍微复杂一点，但它的一个好处是可以避免需要实现基类中所有的虚方法，因为实现所有的基类虚方法有时候是不切实际的或者是效率很低的。

每一个wxEvtHandler的派生类，例如frame，按钮，菜单以及文档等，都会在其内部维护一个事件表，用来告诉wxWidgets事件和事件处理过程的对应关系。所有继承自wxWindow的窗口类，以及应用程序类都是wxEvtHandler的派生类.

要创建一个静态的事件表(意味着它是在编译期间创建的)，你需要下面几个步骤：

1. 定义一个直接或者间接继承自wxEvtHandler的类.
2. 为每一个你想要处理的事件定义一个处理函数。
3. 在这个类中使用DECLARE_EVENT_TABLE声明事件表。
4. 在.cpp文件中使用使用BEGIN_EVENT_TABLE和END_EVENT_TABLE实现一个事件表。
5. 在事件表的实现中增加事件宏，来实现从事件到事件处理过程的映射。

所有的事件处理函数拥有相同的形式。他们的返回值都是void，他们都不是虚函数，他们都只有一个事件对象作为参数。（如果你熟悉MFC，这可能会让你觉得轻松，因为在MFC中消息处理函数并没有一个统一的形式。）这个事件对象的类型是随这个处理函数要处理的事件的变化而变化的。例如简单控件（比如按钮）的命令处理函数和菜单命令的处理函数的参数都是wxCommandEvent类型，而size事件（这个事件通常是由用户改变窗口的客户区尺寸而引起的）处理函数的参数则是wxSizeEvent的类型。不同的事件参数类型可以调用的方法也不相同，通过这些方法，你可以获得事件产生的原因以及产生这个事件的控件的值的改变情况（比如，文本框中的值的改变）。当然最简单的情形是你完全不需要访问这个参数的任何方法，比如按钮点击事件。

让我们来扩展一下前一章中的例子，来增加一个窗口大小改变事件的处理和一个确定按钮的处理。下面是扩展以后的MyFrame的定义：

```
class MyFrame : public wxFrame
{
public:
    MyFrame(const wxString& title);
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);
    void OnSize(wxSizeEvent& event);
    void OnButtonOK(wxCommandEvent& event);
private:
    DECLARE_EVENT_TABLE()
};
```

增加菜单项的代码和前一章的代码类似，而在frame窗口增加一个按钮的代码也只需要在MyFrame的构造函数中增加下面的代码：

```
wxButton* button = new wxButton(this, wxID_OK, wxT("OK"),
                                   wxPoint(200, 200));
```

类似的，在事件表中也需要相应的增加事件映射宏：

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU     (wxID_ABOUT,     MyFrame::OnAbout)
    EVT_MENU     (wxID_EXIT,      MyFrame::OnQuit)
    EVT_SIZE     (                MyFrame::OnSize)
    EVT_BUTTON   (wxID_OK,        MyFrame::OnButtonOK)
END_EVENT_TABLE()
```

当用户点击关于菜单或者退出菜单的时候，这个事件被发送到frame窗口，而MyFrame的事件表告诉wxWidgets，对于标识符为 wxID_ABOUT的菜单事件应该发送到MyFrame::OnAbout函数，而标识符为wxID_EXIT的菜单事件应该发送到MyFrame:: OnQuit函数。换句话说：当事件处理循环处理这两个事件的时候，相应的函数将会以一个的wxCommandEvent类型的参数被调用。

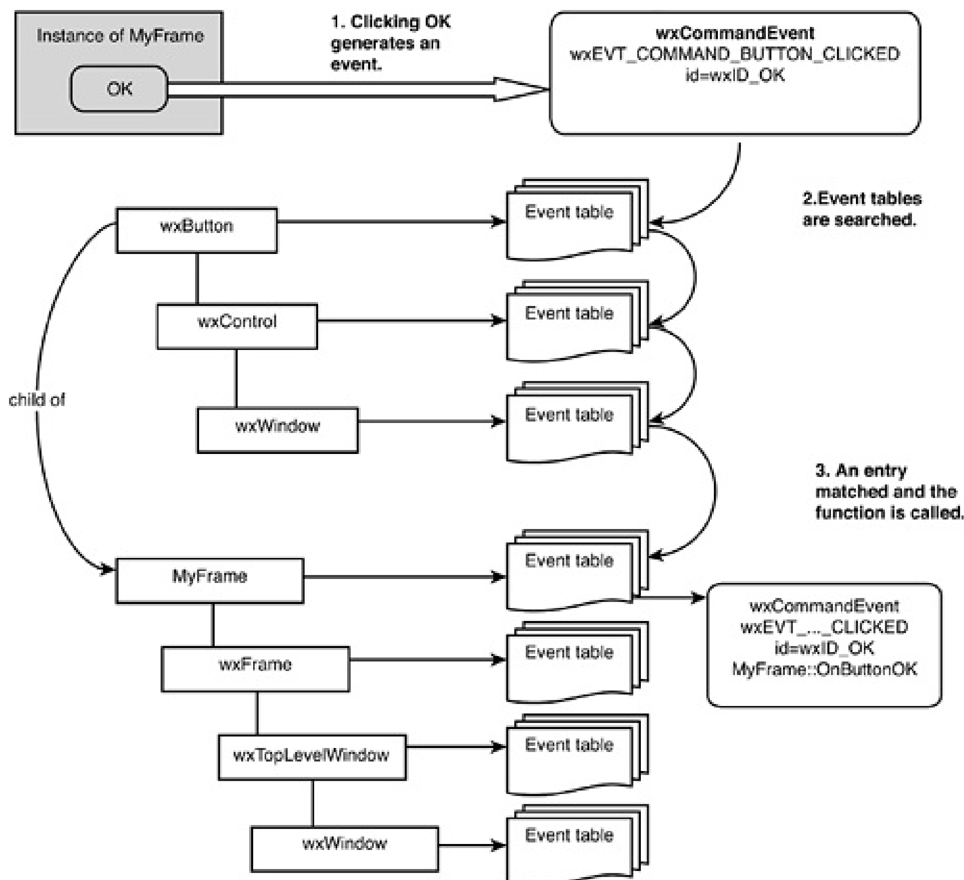
EVT_SIZE事件映射宏不需要标识符参数因为这个事件只会被产生这个事件的控件所处理。

EVT_BUTTON这一行将导致当frame窗口及其子窗口中标识符为wxID_OK的按钮被点击的时候，OnButtonOK函数被调用。这个例子表明，事件可以不必被产生这个事件的控件所处理。让我们假定这个按钮是MyFrame的子窗口。当按钮被点击的时候，wxWidgets会首先检查wxButton类是否指定了相应的处理函数，如果找不到，则在其父亲的类所属的事件表中进行查找，在这个例子中，按钮的父亲是MyFrame类型的一个实例，在其事件表中指明了一个对应的处理函数，因此MyFrame::OnButtonOK函数就被调用了。类似的搜索过程不光发生在窗口控件的父子继承树中，也发生在普通的类继承关系中，这意味着你可以选择在哪里定义事件的处理函数。举例来说，如果你设计了一个对话框，这个对话框需要响应的类似标识符为wxID_OK的command事件。但是你可能需要把这个控件的创建工作留给使用你的代码的其他的程序员，只要他在创建这个控件的时候使用同样的标识符，你仍然可以给这个控件为这个事件定义默认的处理函数。

下图中演示了一次普通的按钮点击事件发生以后，wxWidgets搜索所有事件表的顺序。图中只演示了wxButton和 MyFrame两次继承关系。当用户点击了确认按钮的时候，一个新的wxCommandEvent事件被创建，其中包含标识符wxID_OK和事件类型wxEVT_COMMAND_BUTTON_CLICKED，然后这个按钮的事件表开始通过wxEvtHandler::ProcessEvent函数进行匹配，事件表中的每一个条目都会去尝试匹配，然后是其父类wxControl的事件表，然后是wxWindow的。如果都没有匹配到， wxWidgets会搜索其父亲的类事件表，然后就找到了一条匹配条目：

```
EVT_BUTTON(wxID_OK,MyFrame::OnButtonOK)
```

因此MyFrame::OnButtonOK被调用了。



需要注意的事：只有Command事件(其事件类型直接或者间接的继承自wxCommandEvent)才会被递归的应用到其父亲的事件表。通常这是wxWidgets的用户经常会感到困惑的地方，因此我们把那些不会传递给其父亲的事件表的事件列举如下：wxActivate, wxCloseEvent, wxEraseEvent, wxFocusEvent, wxKeyEvent, wxIdleEvent, wxInitDialogEvent, wxJoystickEvent, wxMenuEvent, wxMouseEvent, wxMoveEvent, wxPaintEvent, wxQueryLayoutInfoEvent, wxSizeEvent, wxScrollWinEvent, 和 wxSysColourChangedEvent，这些事件都不会传给事件源控件的父亲.

这些事件不会传递给其父亲，是因为这些事件仅对产生这个事件的窗口才有意义，举例来说，把一个子窗口的重绘事件发送给它的父亲，其实是没有任何意义的。

3.3 过滤某个事件

wxWidgets事件处理系统实现了一些和C++中的虚方法非常类似的机制，通过这种机制，你可以通过重载某种基类的事件表的方法来改变基类的默认的事件处理过程。在多数情况下，通过这种方法，你甚至可以改变本地原生控件的默认行为。举例来说，你可以过滤某些按键事件以便本地原生的编辑框控件不处理这些按键。要达到这个目的，你需要实现一个继承自wxTextCtrl的新的类，然后在其事件表中使用EVT_KEY_DOWN事件映射宏。过滤所有的按键事件也许不是你想要的，这时候，你可以通过调用wxEvent::Skip函数来提示事件处理过程对于其中的某些按键事件应该继续寻找其父类的事件表。

总的来说，在wxWidgets中，你应该通过调用事件的Skip方法，而不是通过显式直接调用其父类对应函数的方法来实现对特殊事件的过滤。

下面的这个例子演示怎样让你自己的文本框控件只接受"a"到"z"和"A"到"Z"的按键，而忽略其它按键的方法：

```
void MyTextCtrl::OnChar(wxKeyEvent& event)
{
    if ( wxIsalpha( event.KeyCode() ) )
    {
       //这些按键在可以接受的范围，所以按照正常的流程处理
       event.Skip();
    }
    else
    {
       // 这些事件不在我们可以接受的范围，所以我们不调用Skip函数
       // 由于事件表已经匹配并且没有调用Skip函数，所以事件处理过程不会
       // 再继续匹配别的事件表，而是认为事件处理已经结束
       wxBell();
    }
}
```

3.4 挂载事件表

你并不一定非要实现继承自某个类的新类，才可以改变它的事件表。对于那些继承自wxWindow的类来说，有另外一种可取代的方法。你可以通过实现一个新的直接继承自wxEvtHandler的新类，然后定义这个新类事件表，然后使用wxWindow::PushEventHandler函数将这个事件表压入到某个窗口类的事件表栈中。最后压入的那个事件表在事件匹配过程中将会被最先匹配，如果在其中没有匹配到对应的事件处理过程，那么栈中以前的事件表仍将被匹配，如此类推。你还可以用wxWindow::PopEventHandler函数来弹出最顶层的事件表，如果你给wxWindow:: PopEventHandler函数传递的是True的参数，那么这个弹出的事件表将被删除。

这种方法可以避免大量的类重载，也使不同的类的实例共享同一个事件表成为可能。

有时候，你需要手动调用窗口类的事件表，这时候你应该使用wxWindow::GetEventHandler方法，而不是直接使用调用这个窗口类的成员函数。虽然wxWindow::GetEventHandler通常返回这个窗口类本身。但是如果你之前曾经使用 PushEventHandler压入另外一个事件表，那么函数将会返回处于最顶层的事件表。因此使用wxWindow:: GetEventHandler函数才可以保证事件被正确的处理。

PushEventHandler的方法通常用来临时的或者永久的改变图形界面的行为。举例来说，加入你想在你的应用程序实现对话框编辑的功能。你可以捕获这个对话框和它的内部控件的所有的鼠标事件，先使用你自己的事件表处理这些事件，来进行类似拖拽控件，改变控件大小以及移动控件动作。这在联机教学中也是很有用技术。你可以在你自己的事件表中过滤收到的事件，如果是可以接受的，则调用wxEvent::Skip函数正常处理。

3.5 动态事件处理方法

前面我们讨论的事件处理方法，都是静态的事件表，这也是我们处理事件最常用的方式。接下来，我们来讨论一下事件表的动态处理，也就是说在运行期改变事件表的映射关系。使用这种事件映射方法的原因，可能是你想在程序运行的不同时刻使用不同的映射关系，或者因为你使用的那种语言(例如python)不支持静态映射，或者仅仅是因为你更喜欢动态映射。因为动态映射的方法可以使你更精确的控制事件表的细节，你甚至可以单独的将事件表中的某一个条目在运行期打开或者关闭，而前面说的PushEventHandler和PopEventHandler的方法只能针对整个事件表进行处理。除此以外，动态事件处理还允许你在不同的类之间共享事件函数。

和动态事件处理相关的API有两个:wxEvtHandler::Connect和wxEvtHandler::Disconnect。大多数情况下你不需要手动调用wxEvtHandler::Disconnect函数，这个函数将在窗口类被释放的时候自动。

下面我们还用前面的MyFrame类来举例说明：

```
class MyFrame : public wxFrame
{
public:
    MyFrame(const wxString& title);
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);
private:
};
```

你可能已经注意到，这次我们没有使用DECLARE_EVENT_TABLE来声明一个事件表。为了动态进行事件映射，我们需要在OnInit函数中增加下面的代码：

```
frame->Connect( wxID_EXIT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventHandler(MyFrame::OnQuit) );
frame->Connect( wxID_ABOUT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventHandler(MyFrame::OnAbout) );
```

我们传递给Connect函数的三个参数分别为菜单标识符，事件标识符和事件处理函数指针。要注意这里的事件标识符 wxEVT_COMMAND_MENU_SELECTED不同于前面在静态事件表中用于表示事件映射的宏EVT_MENU,实际上EVT_MENU内部也使用了wxEVT_COMMAND_MENU_SELECTED.EVT_MENU其实也自动包含了用于对事件处理指针类型强制转换的宏 wxCommandEventHandler()。一般说来，如果事件处理函数的参数类型是wxXYZEvent,那么其处理函数的类型就应该用 wxXYZEventHandler宏进行强制转换.

如果我们希望在某个时候中止上面的事件映射，可以使用下面的方法：

```
frame->Disconnect( wxID_EXIT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventHandler(MyFrame::OnQuit) );
frame->Disconnect( wxID_ABOUT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventFunction(MyFrame::OnAbout) );
```

3.6 窗口标识符

窗口标识符是在事件系统中用来唯一确定窗口的整数。事实上，在整个应用程序的范围内，窗口标识符不必一定是唯一的，而只要在某个固定的上下文(比如说，在一个frame窗口和它的所有子窗口)内是唯一的就可以了。举例来说：你可以在无数个对话框中使用wxID_OK这个标识符，只要在某个对话框内不要重复使用就可以了。

如果在窗口的构造函数中使用wxID_ANY作为其标识符，则意味着你希望wxWidgets自动为你生成一个标识符。这或者是因为你不关心这个标识符的值，或者是因为这个窗口不需要处理任何事件，或者是因为你将在同一个地方处理所有的事件。如果是最后一种情况，在使用wxEvtHandler::Connect函数或者在静态事件表中，你应该使用wxID_ANY作为窗口的标识符。wxWidgets自动创建的标识符是总是一个负数，所以永远不会和用户定义的窗口标识符重复，用户定义的窗口标识符只能是正整数。

下表列举了wxWidgets提供的一些标准的标识符。你应该尽可能的使用这些标识符，这是由于下面一些原因。某些系统会给特定的标识符提供一些小图片(例如GTK+系统上的OK和取消按钮)或者提供默认的处理函数(例如自动产生wxID_CANCEL事件来响应Escape键)。在 Mac OS X系统上，wxID_ABOUT, wxID_PREFERENCES和wxID_EXIT菜单项也有特别的处理。另外一些wxWidgets的控件也会自动处理标识符为 wxID_COPY, wxID_PASTE或wxID_UNDO等的一些菜单或者按钮的命令。

标识符名称	描述
wxID_ANY	让wxWidgets自动产生一个标识符
wxID_LOWEST	最小的系统标识符值 (4999)
wxID_HIGHEST	最大的系统标识符值 (5999)
wxID_OPEN	打开文件
wxID_CLOSE	关闭窗口
wxID_NEW	新建窗口文件或者文档
wxID_SAVE	保存文件
wxID_SAVEAS	文件另存为(应该弹出文件位置对话框)
wxID_REVERT	恢复文件在磁盘上的状态
wxID_EXIT	退出应用程序
wxID_UNDO	撤消最近一次操作
wxID_REDO	重复最近一次操作
wxID_HELP	帮助 (例如对话框上的帮助按钮可以用这个标识符)

wxID_PRINT_SETUP	打印设置
wxID_PREVIEW	打印预览
wxID_ABOUT	显示一个用来描述整个程序的对话框
wxID_HELP_CONTENTS	显示上下文帮助
wxID_HELP_COMMANDS	显示应用程序命令
wxID_HELP_PROCEDURES	显示应用程序过程
wxID_HELP_CONTEXT	未使用
wxID_CUT	剪切
wxID_COPY	复制到剪贴板
wxID_PASTE	粘贴
wxID_CLEAR	清除
wxID_FIND	查找
wxID_DUPLICATE	复制
wxID_SELECTALL	全选
wxID_DELETE	删除
wxID_REPLACE	覆盖
wxID_REPLACE_ALL	全部覆盖
wxID_PROPERTIES	查看属性
wxID_VIEW_DETAILS	列表框中的按照详细信息方式显示
wxID_VIEW_LARGEICONS	列表框按照大图标的方式显示
wxID_VIEW_SMALLICONS	列表框中按照小图标的方式显示
wxID_VIEW_LIST	列表框中按照列表的的方式显示
wxID_VIEW_SORTDATE	按照日期排序
wxID_VIEW_SORTNAME	按照名称排序
wxID_VIEW_SORTSIZE	按照大小排序
wxID_VIEW_SORTTYPE	按照类型排序
wxID_FILE1 to wxID_FILE9	显示最近使用的文件
wxID_OK	确定
wxID_CANCEL	取消
wxID_APPLY	应用变更
wxID_YES	YES

wxID_NO	No
wxID_STATIC	静态文本或者静态图片可以用这个标识符
wxID_FORWARD	向前
wxID_BACKWARD	向后
wxID_DEFAULT	恢复默认设置
wxID_MORE	显示更多选项
wxID_SETUP	显示一个设置对话框
wxID_RESET	重置所有选项
wxID_CONTEXT_HELP	显示上下文帮助
wxID_YESTOALL	全部选是
wxID_NOTOALL	全部选否
wxID_ABORT	中止当前操作
wxID_RETRY	重试
wxID_IGNORE	忽略错误
wxID_UP	向上
wxID_DOWN	向下
wxID_HOME	首页
wxID_REFRESH	刷新
wxID_STOP	停止正在进行的操作
wxID_INDEX	显示一个索引
wxID_BOLD	加粗显示
wxID_ITALIC	斜体显示
wxID_JUSTIFY_CENTER	居中
wxID_JUSTIFY_FILL	格式
wxID_JUSTIFY_RIGHT	右对齐
wxID_JUSTIFY_LEFT	左对齐
wxID_UNDERLINE	下划线
wxID_INDENT	缩进
wxID_UNINDENT	反缩进
wxID_ZOOM_100	放大到100%
wxID_ZOOM_FIT	缩放到整页
wxID_ZOOM_IN	放大

wxID_ZOOM_OUT	缩小
wxID_UNDELETE	反删除
wxID_REVERT_TO_SAVED	恢复到上次保存的状态

为了避免你自己定义的标识符和这些预定义的标识符重复，你可以使用大于wxID_HIGHEST的标识符或者小于wxID_LOWEST的标识符。

3.7 自定义事件

如果你要使用自定义的事件，你需要下面的几个步骤:

1. 从一个合适的事件类派生一个你自己的事件类，声明动态类型信息并且实现一个Clone函数，按照你自己的意愿增加新的数据成员和函数成员.如果你希望这个事件在窗口继承关系之间传递，你应该使用的wxCommandEvent派生类，如果你希望这个事件的处理函数可以调用Veto(译者注:某些事件可以调用这个函数来阻止后续可能对这个事件进行的任何操作(如果有的话)，最典型的例子是关闭窗口事件wxEVT_CLOSE)，你应该使用wxNotifyEvent的派生类.
2. 为这个事件类的处理函数定义类型.
3. 定义一个你的事件类支持的事件类型的表。这个表应该定义在你的头文件中。用BEGIN_DECLARE_EVENT_TYPES()宏和END_DECLARE_EVENT_TYPES()宏包含起来。其中的每一个支持的事件的声明应该使用DECLARE_EVENT_TABLE (name, integer)格式的宏.然后在你的.cpp文件中使用DEFINE_EVENT_TYPE(name)来实现这个事件类.
4. 为每个你的事件类支持的事件定义一个事件映射宏。

我们还是通过例子来让上面这段绕口的话显的更生动一些。假如我们要实现一个新的控件wxFontSelectorCtrl,这个控件将可以显示字体的预览。用户通过点击字体的预览来弹出一个对话框让用户可以更改字体。应用程序也许想拦截这个字体改变事件，因此我们在我们的底层鼠标消息处理过程中将会给应用程序发送一个自定义的字体改变事件。

因此我们需要定义一个新的事件wxFontSelectorCtrlEvent.应用程序可以通过事件映射宏EVT_FONT_SELECTION_CHANGED(id, func)来增加对这个事件的处理。我们还需要给这个事件定义一个事件类型: wxEVT_COMMAND_FONT_SELECTION_CHANGED. 这样，我们的头文件看上去就象下面的样子:

```
/*!
 * Font selector event class
 */
class wxFontSelectorCtrlEvent : public wxNotifyEvent
{
public:
    wxFontSelectorCtrlEvent(wxEventType commandType = wxEVT_NULL,
      int id = 0): wxNotifyEvent(commandType, id)
    {}
    wxFontSelectorCtrlEvent(const wxFontSelectorCtrlEvent& event): wxNotifyEvent(event)
    {}
    virtual wxEvent *Clone() const
                  { return new wxFontSelectorCtrlEvent(*this); }
DECLARE_DYNAMIC_CLASS(wxFontSelectorCtrlEvent);
};
typedef void (wxEvtHandler::*wxFontSelectorCtrlEventFunction)
                                        (wxFontSelectorCtrlEvent&);
/*!
 * Font selector control events and macros for handling them
 */
BEGIN_DECLARE_EVENT_TYPES()
    DECLARE_EVENT_TYPE(wxEVT_COMMAND_FONT_SELECTION_CHANGED, 801)
END_DECLARE_EVENT_TYPES()
#define EVT_FONT_SELECTION_CHANGED(id, fn) DECLARE_EVENT_TABLE_ENTRY(
 wxEVT_COMMAND_FONT_SELECTION_CHANGED, id, -1, (wxObjectEventFunction) (wxEventFunction)
(wxFontSelectorCtrlEventFunction) & fn,
(wxObject *) NULL ),
```

而在我们的.cpp文件中，看上去则象下面的样子:

```
DEFINE_EVENT_TYPE(wxEVT_COMMAND_FONT_SELECTION_CHANGED)
IMPLEMENT_DYNAMIC_CLASS(wxFontSelectorCtrlEvent, wxNotifyEvent)
```

然后，在我们的新控件的鼠标处理函数中，可以通过下面的方法在检测到用户选择了一个新的字体的时候，发送一个自定义的事件：

```
wxFontSelectorCtrlEvent event(
                    wxEVT_COMMAND_FONT_SELECTION_CHANGED, GetId());
event.SetEventObject(this);
GetEventHandler()->ProcessEvent(event);
```

现在，在使用我们的新控件的应用程序的代码中就可以通过下面代码来处理我们定义的这个新事件了:

```
BEGIN_EVENT_TABLE(MyDialog, wxDialog)
  EVT_FONT_SELECTION_CHANGED(ID_FONTSEL, MyDialog::OnChangeFont)
END_EVENT_TABLE()
void MyDialog::OnChangeFont(wxFontSelectorCtrlEvent& event)
{
    // 字体已经更改了，可以作一些必要的处理。
    ...
}
```

上面用到的事件标识符801在最新的版本中已经没有用处了，之所以这样写只是为了兼容wxWidgets2.4的版本。

让我们再来看一眼事件映射宏的定义：

```
#define EVT_FONT_SELECTION_CHANGED(id, fn) DECLARE_EVENT_TABLE_ENTRY(
 wxEVT_COMMAND_FONT_SELECTION_CHANGED, id, -1, (wxObjectEventFunction) (wxEventFunction)
(wxFontSelectorCtrlEventFunction) & fn,
(wxObject *) NULL ),
```

这个宏的作用其实是把一个五元组放入到一个数组中，所以这段代码的语法看上去是奇怪了一些，这个五元组的意义分别解释如下：

- 事件类型:一个事件类可以处理多种事件类型，但是在我们的例子中，只处理了一个事件类型，所以就只有一个事件映射宏的记录。这个事件类型必须要和事件处理函数所有处理的事件的类型一致。
- 传递给事件映射宏的标识符:只有当事件的标识符和事件表中的标识符一致的时候，相应的事件处理函数才会被调用。
- 第二标识符。在这里-1表示没有第二标识符。
- 事件处理函数。如果没有类型的强制转换，一些编译器会报错，这也就是我们要定义事件处理函数类型的原因。
- 用户数据，通常都是NULL;

随书附带光盘中的examples/chap03目录中有一个完整的自定义事件的例子，其中包括了一个字体选择控件和一个简单验证类，你可以在你的应用程序中使用她们。你还可以阅读你的wxWidgets发行版目录中的include/wx/event.h来获得更多的灵感。

第三章小结

在这一章里，我们讨论了wxWidgets中的事件分发机制，以及怎样进行动态事件处理，还谈到了窗口标识符以及怎样使用自定义的事件。更多关于事件处理的内容，你可以参考附录H,"wxWidgets怎样处理事件"以及附录I"事件处理类和宏",那里列举了主要的事件处理的类和宏。你还可以参考大量的 wxWidgets的例子来学习事件的用法,尤其是wxWidgets发行版目录samples/event这个例子。

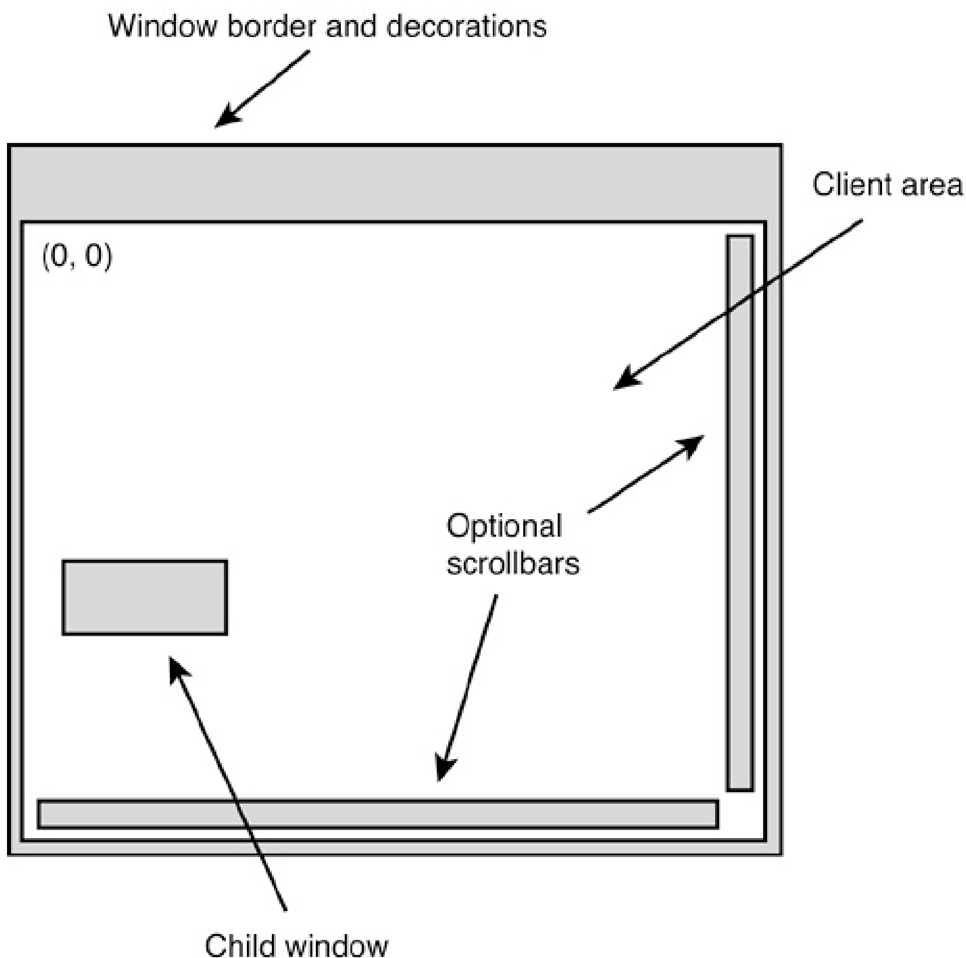
在下一章里，我们将讨论一系列重要的GUI控件以及如何在你的程序中使用这些控件.

第四章 窗口的基础知识

在这一章中，在详细描述应用程序中大量使用的各种窗口类之前，我们首先来看看一个窗口的主要元素。这些要素中的大部分相信你以前在各种平台上的编程经验或者使用经验足以让你对它们非常的熟悉。虽然总会有一些平台相关的差异，但是你会发现，单就功能上来说，各个平台之间有着惊人的相似，而wxWidgets 已经屏蔽了几乎所有这些平台上的差异，还剩下的一小部分差异，主要体现在各个平台可选的窗口类型上的不同。

4.1 窗口解析

你当然大略的知道一个窗口指的是什么，但是为了更好的理解wxWidgets的API,你应该更精通wxWidgets使用的窗口模型的细节。它可能和你在某个特定平台上的窗口概念有些许的不同。下图演示了一个窗口中的各个基本元素：



窗口的概念

一个窗口指的是屏幕上的任何一个拥有以下特征的规则区域：它可以被改变大小，可以自我刷新，可以被显示和隐藏等等。它可以包含别的窗口(比如frame窗口就可以包含菜单条窗口，工具条窗口以及状态条窗口)，也可以子窗口(比如一个静态的文本或者一副静态图片)。通常你在使用wxWidgets编写的程序运行的屏幕上看到的窗口，都和一个wxWindow类或者它的派生类对应，当然也不总是这样。比如本地原生下拉框就不总是用wxWindow建模的。

客户区和非客户区

当我们谈到窗口的大小，我们通常指的是它整个的大小，包括一些用于修饰的边框和标题栏等。而当我们谈到一个窗口的客户区大小，通常都只意味着窗口里面那些能被绘制或者它的子窗口能被放置的位置的大小。例如一个frame窗口的客户区大小就不包括那些菜单栏，状态栏和工具栏所占用的地方。

滚动条

大多数窗口都有显示滚动条的能力，这些滚动条通常是窗口自己增加的而不是有应用程序手动增加的。在这种情况下，客户区的大小还应该减去滚动条所占用的空间。为了优化性能，只有那些拥有wxHSCROLL和wxVSCROLL类型的窗口才会自动生成它们自己的滚动条。关于滚动条更多的情形我们会在本章稍候讨论wxScrolledWindow的时候讨论。

光标和鼠标指针

一个窗口可以拥有一个光标(wxCaret,用来显示当前的文本位置)和一个鼠标指针(wxCursor,用来显示当前鼠标指针的位置).当鼠标移入某个窗口时，wxWidgets会自动显示这个窗口的鼠标指针。当一个窗口变为当前焦点窗口时，如果可以的话，光标将会显示在当前文本的插入位置，或者如果这个焦点是由于鼠标点击造成的，光标将会显示在鼠标对应的位置。

顶层窗口

窗口通常分为象wxFrame,wxDialog,wxPopup这样的顶层窗口和其它窗口。只有顶层窗口创建的时候可以使用NULL作为其父窗口，也指有顶层窗口是延迟删除的(所谓延迟删除的意思是说，它们只有在系统空闲的时候才会被删除，也就是说只有当所有的事件都被处理完以后才会被删除)。而且出了Popup窗口以外，顶层窗口通常都有一个标题栏和一个关闭按钮，只要应用程序允许，它们就可以被拽着满屏幕的跑或者被改变大小。

座标体系

窗口的座标体系通常是左上角为原点(0,0),单位是象素。在某个用于窗口绘制的特定的上下文中，原点和比例允许被改变。这方面详细的可以参考第5章,"窗口绘制和打印".

窗口绘制

当一个窗口需要重绘的时候，它将收到两个事件，wxEVT_ERASE_BACKGROUND事件用于通知应用程序重新绘制背景， wxEVT_PAINT则用于通知重新绘制前景。那些已经准备好使用的窗口空间比如wxButton(按钮)通常知道怎么处理这两个事件，但是如果你是要创建自己的窗口控件，你就需要自己处理这两个事件了。通过获取窗口的变动区域你可以优化你的绘制代码。

颜色和字体

每一个窗口都有一个前景色和一个背景色。默认的背景擦除函数会使用背景色来清除窗口背景，如果没有设置背景色，则会使用当前的系统皮肤推荐的颜色进行背景的清除。前景色则相对来说很少被用到。每一个窗口也拥有一个字体设置，是否用到这个字体设置要取决于这个窗口本身的类型。

窗口变体

在Mac OS X上，一个窗口有一个窗口变体的概念，通过这个概念窗口可以被以不同级别的大小显示：wxWINDOW_VARIANT_NORMAL(默认显示级别), wxWINDOW_VARIANT_SMALL, wxWINDOW_VARIANT_MINI, 或者 wxWINDOW_VARIANT_LARGE.当你有很多信息要展示而屏幕空间不够的时候，你可以使用相对较小的级别，但是这个显示级别的使用应该适可而止。有些程序总是喜欢使用小的显示级别。

改变大小

当一个窗口的大小，无论是来自用户还是应用程序本身的原因，发生变化时，它将收到一个wxEVT_SIZE事件。如果这个窗口拥有子窗口，它们可能需要被重新放置和重新计算大小。处理这种情况推荐的方法是使用sizer类，关于这个类的细节将在第7章，"使用Sizer确定窗口的布局" 这一章详细介绍。大多数已经确定的窗口类都有一个默认的大小和位置，这需要你在创建这些窗口的时候使用wxDefaultSize和 wxDefaultPosition这两个特殊的值。到目前为止，每一个控件都实现了DoGetBestSize函数，这个函数会返回一个基于控件的内容，当前字体以及其它各方面来说最合理的大小。

输入

任何窗口在任何时候都可以接收到鼠标事件，除非某个窗口已经临时捕获了鼠标或者这个窗口已经被禁止使用了，而对于键盘事件来说，只有当前处于活动状态的窗口才可以收到。应用程序自己可以设置自己为活动状态，wxWidgets也会在用户点击某个窗口的时候将其设置为活动状态。正变成活动状态的窗口会收到wxEVT_SET_FOCUS事件，而正失去焦点的窗口会收到wxEVT_KILL_FOCUS事件。请参考第6章：处理输入。

空闲事件处理和用户界面更新

所有的窗口(除非特殊声明)都将收到空闲事件wxEVT_IDLE,这个事件是在所有其它的事件都已经被处理完以后发出的。使用EVT_IDLE事件映射宏来处理。更多的信息请参卡第17章,"多线程编程"中的"空闲时间处理"小节.

其中一个特殊的空闲时间操作就是进行用户界面更新，在这个操作中所有的窗口都可以定义一个函数来更新自己的状态。这个函数将会被周期性的在系统空闲时调用。而EVT_UPDATE_UI(id, func)这个宏则通常不需要作什么事情。更多关于用户界面更新的细节请参考第9章,"创建自定义对话框".

窗口的创建和删除

一般来说，窗口都是在堆上使用new方法创建的，第15章，"内存管理，调试和错误检查"这一章对此有详细的描述，也会提到一些例外情况。大多数的窗口类都可以通过两种方法被创建：单步创建和两步创建。比如wxButton的两种构造函数如下:

```
wxButton();
wxButton(wxWindow* parent,
    wxWindowID id,
    const wxString& label = wxEmptyString,
    const wxPoint& pos = wxDefaultPosition,
    const wxSize& size = wxDefaultSize,
    long style = 0,
    const wxValidator& validator = wxDefaultValidator,
    const wxString& name = wxT("button"));
```

下面演示了使用一步创建的方法创建一个wxButton的实例(其中多数参数采用默认值)：

```
wxButton* button  = new wxButton(parent, wxID_OK);
```

除非是frame或者dialog窗口，对于别的窗口，都必须在构造函数中传入一个非空的父窗口。这会自动把这个新窗口作为这个父窗口的子窗口。当父窗口被释放的时候，它的所有的子窗口也将被释放。正象我们前面提到的那样，你必须输入一个自定义的或者系统内建的标识符给这个窗口以便唯一标识这个窗口。你也可以使用wxID_ANY宏让wxWidgets帮你生成一个。你可以传递位置和大小参数给这个窗口，一个校验类(参考第9章)，一个类型 (接下来会提到)，和一个字符串的名字。这个字符串的名字可以是任意的值或者干脆不填也可以。只有在Xt和Motif系统上这个参数才有意义，因为在这些系统上控件是通过它们的名字来区分的,平常情况下则很少用到。

两步创建的意思是说，你先使用默认的构造函数创建一个实例，然后再使用这个实例Create方法实际创建这个对象。Create的参数和前面使用的构造函数的参数完全相同,还是用wxButton作为例子:

```
wxButton* button  = new wxButton;
button->Create(parent, wxID_OK);
```

窗口在你调用Create函数的时候会收到wxEVT_CREATE事件，你可以对这个事件进行进一步的处理。

使用两步创建的原因是什么呢？第一个原因是有时侯你可能想在晚些时候，在真正需要的时候才完整的创建窗口。另外一个原因是你希望在调用 Create函数之前设置窗口的某些属性值。尤其是这些属性值被Create函数使用的时候就显的更有意义。例如你可能想在窗口被创建之前设置 wxWS_EX_VALIDATE_RECURSIVELY扩展类型，而这个类型必须通过SetExtraStyle函数才可以设置。在这种情况下，对某些对话框类而言，validation必须在Create函数被调用之前被初始化。所以如果你需要这个功能，就必须在调用Create之前初始化这个值。

当你创建一个窗口类，或者其它任何非顶层窗口的派生类的时候，如果它的父窗口是可见的，那么它也总是可见的，你可以通过Show(false)来使它不可见。这和wxDialog或者wxFrame这样的顶层窗口是不一样的。顶层窗口在创建时通常是不可见的，这样可以避免绘

制那些子窗口和排列子控件的时候发生闪烁。你需要通过Show或者(对于模式对话框来说)ShowModal的调用让它可见。

窗口是通过调用其Destroy函数(对于顶层窗口来说)或者delete函数(对于其子窗口来说)来释放的。wxEVT_DESTROY事件会在窗口刚刚要被释放之前被调用。实际上，子窗口是被自动释放的，所以delete函数是很少直接被手动调用的。

窗口类型

窗口拥有一个类型和一个扩展类型。窗口类型是设置窗口创建时的行为和外观的一种简洁的方法。这些类型的值被设置成可以使用类似比特位的方法操作，例如下面的例子:

```
wxCAPTION | wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxTHICK_FRAME
```

wxWindow类有一组基本的类型值，例如边框的类型等，每一个派生类可以增加它们自己的类型。需要特别指出的是，扩展类型的值是不可以拿来给类型用的。

4.2 窗口类概览

在接下来的章节中，我们会介绍最常用的那些窗口类以便你可以在你的应用程序中使用它们。然而如果你是第一次阅读这本书，你可以直接跳到第5章阅读后面的内容，而在晚些时候你需要使用的时候再回过头来阅读本章的内容。

为了让你先大致浏览一下本章的内容，我们把本章将会讨论的窗口类列举如下。对于其他一些窗口类，请参考第12章,"高级窗口类"以及附录E,"wxWidgets中的第三方工具".

基本窗口类

下面的这些基本的窗口类实现了一些最基本的功能。这些类主要是用来作为别的类型的基类以生成更实用的派生类。

- wxWindow. 这是所有窗口类的基类。
- wxControl. 所有控件(比如wxButton)的基类.
- wxControlWithItems. 是那些拥有多个子项目的控件的基类.

顶层窗口类

顶层窗口类通常指那些独立的位于桌面上的类。

- wxFrame. 一个可以包含其他窗口，并且大小可变的窗口类。
- wxMDIParentFrame. 是一个可以管理其他Frame类的类.
- wxMDIChildFrame. 是一个可以被其父窗口管理的frame类.
- wxDialog. 是一种可变大小的用于给用户提供选项的窗口类.
- wxPopupWindow. 是一种暂态的只有很少修饰的顶层窗口.

容器窗口类

容器窗口类可以管理其他窗口

- wxPanel. 这是一个给其它窗口提供布局的窗口.
- wxNotebook. 可以实用TAB页面进行切换的窗口.
- wxScrolledWindow. 可以有滚动条的窗口.
- wxSplitterWindow. 可以管理两个子窗口的一种特殊窗口类.

非静态控件窗口类

这些控件是用户可以操作或者编辑的。

- wxButton. 一种拥有一个标签的按钮控件.
- wxBitmapButton. 一种拥有图片和标签的按钮控件.
- wxChoice. 拥有一个下拉列表的选择控件.
- wxComboBox. 拥有一组选项的可编辑的选择控件.

- wxCheckBox. 拥有一个复选框的控件，复选框有选中和未选中两种状态.
- wxListBox. 拥有一组可选择的字符串项目的列表框.
- wxRadioBox. 拥有一组选项的单选框.
- wxRadioButton. 单选框.
- wxScrollBar. 滚动条控件。
- wxSpinButton. 一个拥有增加和减小两个选项的按钮.
- wxSpinCtrl. 拥有一个文本编辑框和一个wxSpinButton用来编辑整数.
- wxSlider. 这个控件用来在一个固定的范围内选择一个整数.
- wxTextCtrl. 单行或者多行的文本编辑框.
- wxToggleButton. 两态按钮.

静态控件

这些控件提供不能被最终用户编辑的静态信息

- wxGauge. 用来显式数量的控件.
- wxStaticText. 文字标签控件.
- wxStaticBitmap. 用来显示一幅静态图片.
- wxStaticLine. 用来显式静态的一行.
- wxStaticBox. 用来在别的控件周围显示一个静态的方框.

菜单

菜单是一种包含一组命令列表的窗口

控件条

控件条通常在Frame窗口中使用，用来为信息或者命令的访问提供快捷操作

- wxMenuBar. wxFrame上的菜单条.
- wxToolBar. 工具条.
- wxStatusBar. 状态条用来在程序运行过程中显示运行期信息.

4.3 基础窗口类

虽然你不一定有机会直接使用基础窗口类(wxWindow)，但是由于这个类是很多窗口控件的基类，它实现的很多方法在它的子类型中都可以直接拿来使用，所以有必要介绍一下这个基础窗口类。

窗口类wxWindow

wxWindow窗口类既是一个重要的基类，也是一个你可以直接在代码中使用的类。当然，前者使用的频度要比后者大很多。

和前面提到的一样，wxWindow也可以使用单步创建或者两步创建两种方式。单步创建的构造函数原型如下：

```
wxWindow(wxWindow* parent,
    wxWindowID id,
    const wxPoint& pos = wxDefaultPosition,
    const wxSize& size = wxDefaultSize,
    long style = 0,
    const wxString& name = wxT("panel"));
```

可以象下面这样使用:

```
wxWindow* win  = new wxWindow(parent, wxID_ANY,
    wxPoint(100, 100), wxSize(200, 200));
```

窗口类型

每一个窗口类都可以使用定义在下表中的这些窗口基类中的窗口类型。这些类型中不是所有的类些都被所有的控件所支持。例如对于边框的类型。如果在创建窗口的时候你没有指定窗口的边框类型，那么在不同的平台上将会有不同的边框类型的缺省值。在windows平台上，控件边框的缺省值为 wxSUNKEN_BORDER,意为使用当前系统风格的边框。你可以使用类似wxNO_BORDER这样的值来覆盖系统的默认值。

wxSIMPLE_BORDER	在窗口周围显示一个瘦边框.
wxDOUBLE_BORDER	显示一个双层边框.
wxSUNKEN_BORDER	显示一个凹陷的边框，或者使用当前窗口风格设置.
wxRAISED_BORDER	显示一个凸起的边框.
wxSTATIC_BORDER	显示一个适合静态控件的边框. 只支持Windows平台.
wxNO_BORDER	不显示任何边框.
wxTRANSPARENT_WINDOW	定义一个透明窗口 (意思是这个窗口不接收paint事件).只支持windows平台.
wxTAB_TRAVERSAL	使用这个类型允许非Dialog窗口支持使用TAB进行遍历.
wxWANTS_CHARS	使用这个类型来允许窗口接收包括回车和TAB在内的所有的键盘事件。TAB用来在Dialog类型的窗口中遍历各控件。如果没有设置这个类型，这些特殊的按键事件将不会被产生。
wxFULL_REPAINT_ON_RESIZE	在默认情况下，在窗口客户区大小发生改变时，wxWidgets并不会重画整个客户区。设置这个类型将使得wxWidgets改变这种默认的作法，而保持整个客户区的刷新
wxVSCROLL	显示垂直滚动条.
wxHSCROLL	显示水平滚动条.
wxALWAYS_SHOW_SB	如果一个窗口有滚动条，那么在不需要滚动条的时候（当窗口足够大不需要使用滚动条的时候），禁止滚条而不隐藏滚动条。这个类型目前只支持Windows平台和wxWidgets的wxUniversal版本.
wxCLIP_CHILDREN	只支持Windows平台,用于消除由于擦除子窗口的背景而引起的闪烁.

下表列出了窗口的扩展类型，这些扩展类型不能直接和类型混用，而要使用wxWindow::SetExtraStyle函数来进行设置。

wxWS_EX_VALIDATE_RECURSIVELY	在默认情况下，**Validate,transferDataToWindow,**和**transferDataFromWindow**只在窗口的直接子窗口上才可以使用。如果设置了这个扩展类型，那么将可以递归的在各个子窗口上使用。
wxWS_EX_BLOCK_EVENTS	wxCommandEvents事件将会在无法在当前事件表中找到匹配的时候在其父窗口中尝试匹配，设置这个扩展属性可以阻止这个行为。Dialog类型的窗口默认设置了这个类型，但是如果SetExtraStyle被应用程序类调用过的话，默认设置可能被覆盖.
wxWS_EX_TRANSIENT	不要使用这个窗口作为其它窗口的父窗口.这个类型一定只能用于瞬间窗口;否则，如果使用它作为一个dialog或者frame类型窗口的父窗口，如果父窗口在子窗口之前释放，可能导致系统崩溃。
wxWS_EX_PROCESS_IDLE	这个窗口应该处理所有的idle事件，包括那些设置了wxIDLE_PROCESS_SPECIFIED模式的idle事件。
wxWS_EX_PROCESS_UI_UPDATES	这个窗口将处理所有的Ui刷新事件，包括那些设置了wxUPDATE_UI_PROCESS_SPECIFIED的UI刷新事件。参考第9章获得和界面刷新有关的更多的内容.

窗口事件

窗口类和它的派生类可以产生下面的事件（鼠标，键盘和游戏手柄产生的事件将会在第6章描述）。

EVT_WINDOW_CREATE(func)	用于处理**wxEVT_CREATE**事件, 这个事件在窗口刚刚被产生的时候生成, 处理函数的参数类型是**wxWindowCreateEvent.**
EVT_WINDOW_DESTROY(func)	用于处理wxEVT_DELETE事件,在这个窗口即将被删除的时候产生, 处理函数的参数类型是wxWindowDestroyEvent.
EVT_PAINT(func)	用于处理wxEVT_PAINT事件,在窗口需要被刷新的时候产生.处理函数的参数类型是wxPaintEvent.
EVT_ERASE_BACKGROUND(func)	用于处理wxEVT_ERASE_BACKGROUND事件,在窗口背景需要被更新的时候产生. 处理函数的参数是wxEraseEvent.
EVT_MOVE(func)	用于处理wxEVT_MOVE事件, 在窗口移动的时候产生.处理函数的参数类型是wxMoveEvent.
EVT_SIZE(func)	用于处理wxEVT_SIZE事件, 在窗口大小发生变化以后产生.处理函数的参数类型是wxSizeEvent.
EVT_SET_FOCUS(func)EVT_KILL_FOCUS(func)	用于处理wxEVT_SET_FOCUS和wxEVT_KILL_FOCUS事件,在窗口得到或者失去键盘焦点的时候产生. 处理函数参数类型是wxFocusEvent.
EVT_SYS_COLOUR_CHANGED(func)	用于处理wxEVT_SYS_COLOUR_CHANGED事件,当用户在控制面板里更改了系统颜色的时候产生(只支持windows平台). 处理函数参数类型为wxSysColourChangedEvent.
EVT_IDLE(func)	用于处理wxEVT_IDLE事件,在空闲事件产生.处理函数参数类型位wxIdleEvent.
EVT_UPDATE_UI(func)	用于处理wxEVT_UPDATE_UI事件,在系统空闲时间产生用来给窗口一个更新自己的机会.

wxWindow类的成员函数

因为wxWindow类是其它所有窗口类的基类，它拥有很多的成员函数。我们不可能在这里作一一的说明，只能拣其中最重要的一些作简要的说明。不过我们还是推荐你浏览一下wxWidgets手册中的相关部分，以便能够彻底了解wxWindow类提供的所有功能，以及要使用这个功能你需要提供的参数等。

CaptureMouse函数可以捕获所有的鼠标输入（将其限制在本窗口以内）,ReleaseMouse则可以释放前一次的捕获.在绘图程序中，这两个函数是很有用的。它可以让你在鼠标移动到窗口边缘的时候来滚动窗口的客户区，而不是任由鼠标跑到别的窗口并且激活别的窗口。另外的两个函数GetCapture用来获取当前正在使用的捕获设备(如果是被当前的应用程序设置的话),而HasCapture可以用来检测是否鼠标正被这个窗口捕获。

Centre, CentreOnParent和CentreOnScreen三个函数可以让窗口调整自己的位置使其位于屏幕或者是其父窗口的正中间位置。

ClearBackground函数将使用当前的背景色清除当前窗口.

ClientToScreen和ScreenToClient可以将座标在相对于屏幕左上角和相对于客户区左上角之间进行互相转换.

Close函数产生一个wxCloseEvent事件，这个事件通常的处理过程将关闭窗口，释放内存。当然如果应用程序为这个事件定义了特殊的处理函数，那么窗口也有可能不被关闭和释放。

ConvertDialogToPixels和ConvertPixelsToDialog函数可以对数值进行对话框单位和象素单位之间的转换。这在基于字体大小以便应用程序的显示更合理的操作中是很有用的.

Destroy函数将安全的释放窗口.使用这个函数代替直接调用delete操作符因为这个函数下不同的窗口类型的表现是不一样的。对于对话框和frame这样的顶层窗口来说，这个函数会将窗口放入一个等待删除的额窗口列表中，等到这个窗口的所有的事件都处理完毕的时候才会被删除，这可以避免一些事件在已经不存在的窗口上被执行.

Enable允许或者禁止窗口和它的子窗口处理输入事件。在禁止状态下一些窗口会有不同的颜色和外观。Disable函数和Enable函数使用false作为参数的效果是完全一样的。

FindFocus函数是一个静态函数，用它可以找到当前拥有键盘焦点的窗口。

FindWindow函数可以通过标识符或者名字在它的窗口关系树中寻找某个窗口. 返回值可能是它的一个子窗口或者它自己.如果你可以确定你要找的窗口的类型，可以安全的使用wxDynamicCast进行类型强制转换，转换的结果将是一个指向那个类型的指针或者是NULL:

```
MyWindow* window = wxDynamicCast(FindWindow(ID_MYWINDOW), MyWindow);
```

Fit函数会自动改变窗口的大小以便刚好可以容纳它的所有子窗口.这个函数应用被应用在使用基于sizer的布局的时候。FitInside函数是一个类似的函数，区别在于它使用的是虚大小(应用在那些包含滚动条的窗口).

Freeze和Thaw，这两个函数的作用是告诉wxWidgets，在这两个函数之间进行的刷新界面的操作是允许被优化的。举例来说，如果你要在一个文本框控件逐行中加入多行文本，则可以用这个方法来优化显示效果，避免闪烁，这两个函数已经在GTK+版本的wxTextCtrl控件上实现，也适用于 Windows和Max Os X平台的所有类.

GetAcceleratorTable和SetAcceleratorTable用来获取和设置某个窗口的加速键表.

GetBackgroundColour和SetBackgroundColour用来访问窗口的背景颜色属性。这个属性被wxEVT_ERASE_BACKGROUND事件用来绘制窗口背景.如果你更改了背景颜色设置，你应该调用Refresh或者 ClearBackground函数来刷新背景. SetOwnBackgroundColour的作用和SetBackgroundColour基本相同，但是前者不会更改当前窗口的子窗口的背景属性.

GetBackgroundStyle和SetBackgroundStyle用来设置窗口的背景类型. 默认的窗口背景类型是wxBG_STYLE_SYSTEM,它的含义是wxWidgets按照系统默认设置来进行背景绘制。系统默认的背景绘制方法根据控件的不同而不同，比如wxDialog的默认背景绘制方法是什么纹理绘制的方法，而wxListBox则是使用固定颜色的绘制方法。如果你设置了窗口的背景绘制方法类型是wxBG_STYLE_COLOUR,那么wxWidgets会全部用单一颜色的方法绘制背景。而如果你将其设置为 wxBG_STYLE_CUSTOM, wxWidgets将不进行任何的背景绘制工作，你可以自己在擦除背景事件中或者重画事件中自己绘制背景.如果你希望自己绘制背景，请一定设置wxBG_STYLE_CUSTOM为背景类型，否则很容易引起画面的闪烁.

GetBestSize函数以象素为单位返回窗口最合适的大小(因为每个窗口类都需要实现DoGetBestSize函数).这个函数用来提示Sizer系统不要让这个窗口的尺寸太小以致不能正确的显示和使用。举例来说，对于静态文本控件来说，不要让它的字符只能显示一半。对于包含其它子窗口的窗口比如wxPanel来说，这个尺寸等于对这个窗口调用Fit函数以后的尺寸的大小。

GetCaret和SetCaret函数用来访问或者设置窗口的光标.

GetClientSize和SetClientSize用来访问和设置窗口的客户区大小，单位是象素. 客户区大小指的是不包括边框和修饰的区域，或者说你用户可以绘制或者用来放置子窗口的区域.

GetCursor和SetCursor函数用来访问和设置窗口的鼠标指针.

GetDefaultItem函数返回一个指向这个窗口默认的子按钮的指针或者返回NULL。默认子按钮是当用户按回车键的时候默认激活的按钮，可以通过wxButton::SetDefault函数进行设置.

GetDropTarget和SetDropTarget函数用来取得或者设置和窗口关联的wxDropTarget对象，这个对象用来处理和控制窗口的拖放操作. 我们将在第11章"剪贴板和拖放"中详细介绍拖放有关的操作。

GetEventHandler和SetEventHandler函数用来访问和设置窗口的第一事件表.默认情况下，窗口的事件表就是窗口自己，但是你可以指定不同的事件表，也可以通过PushEventHandler和PopEventHandler函数设置一个事件表链。然后让不同的事件表处理不同的事件. wxWidgets将搜索整个事件表链来寻找匹配的事件处理函数处理新收到的事件，详情请参阅第3章，“事件处理”.

GetExtraStyle和SetExtraStyle函数用来获取和设置窗口的扩展类型位。扩展类型宏通常以wxWS*EX*开头.

GetFont和SetFont函数获取和设置和窗口相关的字体. SetOwnFont和SetFont的功能相似,唯一的区别在于前者设置的字体不会被子窗口继承.

GetForegroundColour和SetForegroundColour函数用来操作窗口的前景颜色属性. 和SetOwnForegroundColour函数的区别仅在于是否改变子窗口的这个属性。

GetHelpText和SetHelpText用来获取和设置窗口相关的上下文帮助.这个文本属性实际上存储在当前的wxHelpProvider实现中，而不是存在于窗口类中.

GetId和SetId用来操作窗口标识符.

GetLabel函数返回窗口相关联的标签.具体的含义取决的特定的窗口类.

GetName和SetName用来操作窗口名称,这个名字不需要是唯一的。这个名称对wxWidgets来说没有什么意义,但是在Motif系统中被用于窗口资源的名称.

GetParent返回窗口的父窗口.

GetPosition以象素单位返回相对于父窗口的窗口左上角座标。

GetRect返回一个wxRect对象 (参考第13章, "数据结构和类型"),其中包含了象素单位的这个窗口的大小和位置信息.

GetSize和SetSize函数获取和设置窗口象素单位的窗口长宽.

GetSizer和SetSizer函数用来操作这个窗口绑定的最顶级的窗口布局对象.

GetTextExtent用于返回当前字体下某个字符串的象素长度.

GetToolTip和SetToolTip用来操作这个窗口的tooltip对象.

GetUpdateRegion函数返回窗口自上次Paint事件以来需要刷新的区域.

GetValidator和SetValidator函数用来操作这个窗口可选的wxValidator对象。详情请参考第9章.

GetVirtualSize返回窗口的虚大小，通常就是和滚动条绑定的那个大小.

GetWindowStyle和SetWindowStyle用来操作窗口类型比特位。

InitDialog函数发送一个wxEVT_INIT_DIALOG事件来来开始对话框数据传送.

IsEnabled用来检测当前窗口的使能状态.

IsExposed用来检测一个点或者一个矩形区域是否位于需要刷新的范围。

IsShown用来指示窗口是否可见.

IsTopLevel用来指示窗口是否是顶层窗口(仅用于wxFrame或者wxDialog).

Layout函数用来在窗口已经指定一个布局对象的情况下更新窗口布局。参考第7章.

Lower函数用来将窗口移到窗口树的最低层，而Raise则把窗口移动到窗口树的最顶层.这两个函数既可以用于顶层窗口，也可以用于子窗口.

MakeModal函数禁用其它所有的顶层窗口，以便用户只能和当前这个顶层窗口交互.

Move函数用来移动窗口.

MoveAfterInTabOrder更改窗口的TAB顺序到作为参数的窗口的后面, 而MoveBeforeInTabOrder则将其挪到参数窗口的前面.

PushEventHandler压入一个事件表到当前的事件表栈, PopEventHandler函数弹出并且返回事件表栈最顶层的事件表. RemoveEventHandler则查找事件表栈中的一个事件表并且将其出栈.

PopupMenu函数在某个位置弹出一个菜单.

Refresh和RefreshRect函数导致窗口收到一个重画事件(和一个可选的擦除背景事件).

SetFocus函数让本窗口收到键盘焦点.

SetScrollbar函数用来设置窗口内建的滚动条的属性.

SetSizeHints函数用来定义窗口的最小最大尺寸，依次窗口尺寸增量的大小，仅对顶层窗口适用.

Show函数用来显示和隐藏窗口; Hide函数的作用相当于适用false作为参数调用Show函数.

transferDataFromWindow和transferDataToWindow获取和传输数据到窗口对象，并且在没有被重载的情况下会调用验证函数.

Update立即重画窗口已经过期的区域.

UpdateWindowUI函数发送wxUpdateUIEvents事件到窗口，以便给窗口一个更新窗口元素（比如工具条和菜单）的机会.

Validate使用当前的验证对象验证窗口数据.

wxControl类

wxControl是一个虚类。它继承自wxWindow，用来作为控件的基类: 所谓控件指的是那些可以显示数据项并且通常需要响应鼠标或者键盘事件的那些窗口类.

wxControlWithItems类

wxControlWithItems也是一个虚类，用来作为wxWidgets的一些包含多个数据项的控件（比如wxListBox, wxCheckListBox,wxChoice和wxComboBox等）的基类。使用这个基类的目的为了给这些具有相似功能的控件提供一致的API函数。

wxControlWithItems的数据项拥有一个字符串和一个和这个字符串绑定的可选的客户数据。客户数据可以是两种类型，要么是无类型指针(void*),这意味这这个控件只帮忙存储客户数据但是永远不会使用客户数据。另外一种是有类型（wxClientData）指针,对于后一种情况，客户数据会在控件被释放或者数据项被清除的时候被自动释放。同一个控件的所有数据项必须拥有同样的客户区数据类型：要么是前者，要么是后者。客户区数据的类型是在第一次调用

Append函数或者，SetClientData函数或者SetClientObject函数的时候被确定的。如果要使用有类型指针客户数据，你应该自定义一个继承自wxClientData的类，然后将它的实例指针传递给Append函数或者 SetClientObject函数。

wxControlWithItems的成员函数

Append函数给这个控件增加一个或者一组数据项. 如果是增加一个数据项，你可以用第二个参数指定有类型客户数据指针或者无类型客户数据指针。比如:

```
wxArrayString strArr;
strArr.Add(wxT("First string"));
strArr.Add(wxT("Second string"));
controlA->Append(strArr);
controlA->Append(wxT("Third string"));
controlB->Append(wxT("First string"), (void *) myPtr);
controlC->Append(wxT("First string"), new MyTypedData(1));
```

Clear函数清除控件所有数据项并且清除所有的客户数据.

Delete函数使用基于0的索引清除一个数据项以及它的客户数据。

FindString返回一个和某个字符串基于0的数据项的索引。如果找不到则返回wxNOT_FOUND.

GetClientData和GetClientObject返回某个数据项的客户区数据指针(如果有的话).
SetClientData和SetClientObject则可以用来设置这个指针.

GetCount数据项的总数.

GetSelection返回选中的数据项或wxNOT_FOUND. SetSelection则用来设置某个数据项为选中状态.

GetString用来获取某个数据项的字符串; SetString则用来设置它.

GetStringSelection用来返回选中的数据项的字符串或者一个空的字符串; SetStringSelection则用来设置选中的字符串。你应该先调用FindString函数保证这个字符串是存在于某个数据项的，否则可能引发断言失败.

Insert在控件数据项的某个位置插入一个数据项(可以包含也可以不包含客户数据).

IsEmpty则用来检测一个控件的数据项是否为空.

4.4 顶层窗口

顶层窗口直接被放置在桌面上而不是包含在其它窗口之内。如果应用程序允许，他们可以被移动或者重新改变大小。总共有三种基础的顶层窗口类型。 wxFrame和wxDialog都是从一个虚类wxTopLevelWindow继承来的。另外一个wxPopupWindow功能相对简单，是直接从wxWindow继承过来的。一个dialog既可以是模式的也可以是非模式的，而frame通常都是非模式的。模式对话框的意思是说当这个对话框弹出时，应用程序除了等待用户关闭这个对话框以外不再作别的事情。对于那些要等待用户响应以后才能继续的操作来说。这是比较合适的。但是寻找替代的解决方案通常也不是一件坏事。举例来说，在工具条上设置一个字体选项可能比弹出一个模式的对话框让用户选择字体更能是整个应用程序看上去更流畅。

顶层窗口通常都拥有一个标题栏，这个标题栏上有一些按钮或者菜单或者别的修饰用来关闭，或者最小化，或者恢复这个窗口。而frame窗口则通常还会拥有菜单条，工具条和状态条。但是通常对话框则没有这些。在Mac OS X上，frame窗口的菜单条通常不是显示在frame窗口的顶部而是显示在整个屏幕的顶部。

不要被"顶层窗口"的叫法和wxApp::GetTopWindow函数给搞糊涂了。wxWidgets使用后者来得到应用程序的主窗口，这个主窗口通常是你在应用程序里创建的第一个frame窗口或者dialog窗口。

如果需要，你可以使用全局变量wxTopLevelWindows来访问所有的顶层窗口，这个全局变量是一个wxWindowList类型.

wxFrame

wxFrame是主应用程序窗口的一个通用的选择。下图演示了常见的各个frame窗口元素。frame窗口拥有一个可选的标题栏（上面有一些类似关闭功能的按钮），一个菜单条，一个工具栏，一个状态栏。剩下的区域则称为客户区，如果拥有超过一个子窗口，那么他们的尺寸和位置是由应用程序决定的。你应该通过第7章中会讲到的窗口布局控件来作相关的事情。或者如果只有两个子窗口的话，你可以考虑使用一个分割窗口。后者我们稍后就会讲到。

.tmp)

在某些平台上不允许直接绘制frame窗口，因此你应该增加一个wxPanel作为容器来放置别的子窗口控件，以便可以通过键盘来遍历各个控件。

frame窗口是允许包含多个工具条的，但是它只会自动放置第一个工具条。因此你需要自己明确的布局另外的工具条。

你很应该基于wxFrame实现你自己的frame类而不是直接使用wxFrame类，并且在其构造函数中创建自己的菜单条和子窗口。这会让你更方便处理类似wxEVT_CLOSE事件和别的命令事件。

你可以给frame窗口指定一个图标以便让系统显示在任务栏上或者文件管理器中。在windows平台上，你最好指定一组16x16和 32x32并且拥有不同颜色深度的图标。在linux平台上也一样，windows系统上的图标已经足够。而在Max OsX上，你需要更多的不同颜色和深度的图标。关于图标和图标列表的更多信息，可以参考第10章，ï¿½在应用程序中使用图片ï¿½。

除了默认的构造函数以外，wxFrame还拥有下面的构造函数

```
wxFrame(wxWindow* parent, wxWindowID id, const wxString& title,
    const wxPoint& pos = wxDefaultPosition,
    const wxSize& size = wxDefaultSize,
    long style = wxDEFAULT_FRAME_STYLE,
    const wxString& name = wxT("frame"));
```

例如：

```
wxFrame* frame = new wxFrame(NULL, ID_MYFRAME,
    wxT("Hello wxWidgets"), wxDefaultPosition,
    wxSize(500, 300));
frame->Show(true);
```

注意在你显式的调用Show函数之前，frame是不会被显示的。以便应用程序有机会在所有的子窗口还没有被显示的时候进行窗口布局。

没有必要给这个新建的frame窗口指定父窗口。如果指定了父窗口并且指定了wxFRAME_FLOAT_ON_PARENT类型位，这个新建的窗口将会显示在它的父窗口的上层。

要删除一个frame窗口，应该使用Destroy方法而不是使用delete操作符，因为Destroy会在处理完这个窗口的所有事件以后才真正释放这个窗口。调用Close函数会导致这个窗口收到wxEVT_CLOSE事件，这个事件默认的行为是调用Destroy方法。当一个 frame窗口被释放时。它的所有的子窗口都将被释放，谁叫他们自己不是顶层窗口呢。

当最后一个顶层窗口被释放的时候，应用程序就退出了（这种默认的行为可以通过调用wxApp:: SetExitOnFrameDelete改变）。你应该在你的主窗口的wxEVT_CLSE事件处理函数中调用Destroy释放其它的顶层窗口（比如一个查找对话框之类），否则可能出现主窗口已经关闭但是应用程序却不退出的情况。

frame窗口没有类似dialog窗口那样的wxDialog::ShowModal方法可以禁止其它顶层窗口处理消息，然后，还是可以通过别的方法达到类似的目的。一个方法是通过创建一个wxWindowDisabler对象，另外一个方法是通过wxModalEventLoop对象，传递这个frame窗口的指针作为参数，然后调用Run函数开始一个本地的事件循环，然后调用Exit函数（通常是在这个Frame窗口的某个事件处理函数里）退出这个循环。

下图演示了一个用户应用程序的frame窗口在windows上运行的样子。这个frame窗口拥有一个标题栏，一个菜单条，一个五彩的工具栏，在frame窗口的客户区有个分割窗口，底端有一个状态条，正显示着当用户的鼠标划过工具条上的按钮或者菜单项的时候显示的提示。

.tmp)

wxFrame的窗口类型比特位

Frame窗口比起基本窗口类，增加了下面一些类型：

wxDEFAULT_FRAME_STYLE	其值为 **wxMINIMIZE_BOX | wxMAXIMIZE_BOX | wxRESIZE_BORDER | wxSYSTEM_MENU | wxCAPTION | wxCLOSE_BOX.**
wxICONIZE	以最小化的方式显示窗口。目前只适用于Windows平台.
wxCAPTION	在窗口上显示标题.
wxMINIMIZE	和wxICONIZE的意义相同. 也只适用于Windows平台.
wxMINIMIZE_BOX	显示一个最小化按钮.
wxMAXIMIZE	以最大化方式显示窗口. 仅适用于Windows.
wxMAXIMIZE_BOX	在窗口上显示最大化按钮.
wxCLOSE_BOX	在窗口上显示关闭按钮.
wxSTAY_ON_TOP	这个窗口显示在其它所有顶层窗口之上.仅适用于Windows.
wxSYSTEM_MENU	显示系统菜单.
wxRESIZE_BORDER	边框可改变大小.
wxFRAME_TOOL_WINDOW	窗口的标题栏比正常情况要小，而且在windows平台上，这个窗口不会在任务栏上显示.
wxFRAME_NO_TASKBAR	创建一个标题栏是正常大小，但是不在任务栏显示的窗口。目前支持windows和linux。需要注意在windows平台上，这个窗口最小化时将会最小化到桌面上，这有时候会让用户觉得不习惯，因此这种类型的窗口最好不要使用wxMINIMIZE_BOX类型。
wxFRAME_FLOAT_ON_PARENT	拥有这种类型的窗口总会显示在它的父窗口的上层。使用这种类型的窗口必须拥有非空父窗口，否则可能引发断言错误.
wxFRAME_SHAPED	拥有这种类型的窗口可以通过调用SetShape函数来使其呈现不规则窗口外貌.

下表列出的是frame窗口的扩展类型，需要使用wxWindow::SetExtraStyle函数才可以设置扩展类型

wxFRAME_EX_CONTEXTHELP	在**windows**平台上，这个扩展类型导致标题栏增加一个帮助按钮。当这个按钮被点击以后，窗口会进入一种上下文帮助模式。在这种模式下，任何应用程序被点击时，将会发送一个**wxEVT_HELP**事件。
wxFRAME_EX_METAL	在Mac OS X平台上,这个扩展类型将会导致窗口使用金属外观。请小心使用这个扩展类型，因为它可能会假定你的应用程序用户默认安装类声卡之类的多媒体设备。

wxFrame的事件

下表列出了frame窗口比基本窗口类增加的事件类型：

EVT_ACTIVATE(func)	用来处理**wxEVT_ACTIVATE**事件**,** 在**frame**窗口被激活或者去激活的时候产生**.**处理函数的参数类型为**wxActivateEvent.**
EVT_CLOSE(func)	用来处理wxEVT_CLOSE事件, 在应用程序准备关闭窗口的时候产生. 处理函数的参数类型是wxCloseEvent。这个类型支持Veto函数调用以阻止这个事件的进一步处理.
EVT_ICONIZE(func)	用来处理wxEVT_ICONIZE事件, 当窗口被最小化或者被恢复普通大小的时候产生.处理函数的参数类型为wxIconizeEvent. 通过使用IsIconized函数来判断究竟是最小化事件还是恢复事件.
EVT_MAXIMIZE(func)	用来处理wxEVT_MAXIMIZE事件, 当窗口被最大化或者从最大化恢复的时候产生. 处理函数的参数类型为wxMaximizeEvent. 通过IsMaximized函数判断究竟是最大化还是从最大化状态恢复.

wxFrame的成员函数

下面将介绍wxFrame类的主要的成员函数。因为wxFrame类是从wxTopLevelWindow和wxWindow继承过来的，请同样参考这两个类的成员函数。

CreateStatusBar函数在frame窗口的底部创建一个拥有一个或多个域的状态栏. 可以使用SetStatusText函数来设置状态栏上的文字,使用SetStatusWidths函数来设置状态栏每个域的宽度(参考本章稍后对 wxStatusBar的介绍). 使用举例:

```
frame->CreateStatusBar(2, wxST_SIZEGRIP);
int widths[3] = { 100, 100, -1 };
frame->SetStatusWidths(3, widths);
frame->SetStatusText(wxT("Ready"), 0);
```

CreateToolBar 函数在frame窗口的菜单条下创建一个工具栏. 当然你也可以直接创建一个wxToolBar实例，然后调用wxFrame::SetToolBar函数以便让frame类来管理你刚创建的toolbar.

GetMenuBar用来SetMenuBar操作frame和绑定的菜单条.一个frame窗口只可以有一个菜单条。如果你新创建了一个，那么就的那个将被删除和释放。

GetTitle和SetTitle函数用来访问窗口标题栏上的文本.

Iconize函数使得frame窗口最小化或者从最小化状态恢复.你可以使用IsIconized函数检查当前的最小化状态.

Maximize函数使得frame窗口最大化或者从最大化状态恢复. 你可以用IsMaximized函数来检查当前的最大化状态.

SetIcon函数可以设置在frame窗口最小化的时候显示的图标。各个平台的窗口管理器也可能把这个图标用于别的用途，比如显示在任务条上或者在显示在窗口浏览器中。你还可以通过SetIcons函数指定一组不同颜色和深度的图标列表.

SetShape函数用来给frame窗口设置特定的显示区域。目前支持这个函数的平台有Windows, Mac OS X, 和GTK+以及打开某种X11扩展功能的X11版本.

ShowFullScreen函数使用窗口在全屏和正常状态下切换，所谓全屏状态指的是frame窗口将隐藏尽可能多的修饰元素，而尽可能把客户区以最大化的方式显示。可以使用IsFullScreen函数来检测当前的全屏状态.

不规则的Frame窗口

如果你希望制作一个外貌奇特的应用程序，比如一个时钟程序或者一个媒体播放器，你可以给frame窗口指定一个不规则的区域，只有这个区域范围内的部分才会被显示出来。下图演示了一个没有任何标题栏，状态栏，菜单条的frame窗口，它的重画函数显示了一个企鹅，这个窗口被设置了一个区域使得只有这个企鹅才会显示。

.tmp)

显示一个有着奇特外观的程序的代码并不复杂，你可以在附带光盘的samples/shaped目录里找到一个使用不同图片的完整的例子。总的来说，当窗口被创建时，它加载了一幅图片，并且使用这幅图片的轮廓创建了一个区域。在GTK+版本上，设置窗口区域的动作必须在发送了窗口创建事件之后，所以你需要使用宏定义区别对待GTK+的版本。下面的例子演示了你需要怎样对事件表，窗口的构造函数和窗口的创建事件处理函数进行修改：

```
BEGIN_EVENT_TABLE(ShapedFrame, wxFrame)
    EVT_MOTION(ShapedFrame::OnMouseMove)
    EVT_PAINT(ShapedFrame::OnPaint)
#ifdef __WXGTK__
    EVT_WINDOW_CREATE(ShapedFrame::OnWindowCreate)
#endif
END_EVENT_TABLE()
ShapedFrame::ShapedFrame()
       : wxFrame((wxFrame *)NULL, wxID_ANY, wxEmptyString,
                  wxDefaultPosition, wxSize(250, 300),
                  | wxFRAME_SHAPED
                  | wxSIMPLE_BORDER
                  | wxFRAME_NO_TASKBAR
                  | wxSTAY_ON_TOP
            )
{
    m_hasShape = false;
    m_bmp = wxBitmap(wxT("penguin.png"), wxBITMAP_TYPE_PNG);
    SetSize(wxSize(m_bmp.GetWidth(), m_bmp.GetHeight()));
#ifndef __WXGTK__
    // On wxGTK we can't do this yet because the window hasn't
    // been created yet so we wait until the EVT_WINDOW_CREATE
    // event happens. On wxMSW and wxMac the window has been created
    // at this point so we go ahead and set the shape now.
    SetWindowShape();
#endif
}
// Used on GTK+ only
void ShapedFrame::OnWindowCreate(wxWindowCreateEvent& WXUNUSED(evt))
{
    SetWindowShape();
}
```

为了创建一个区域模板，我们从一幅图片的创建了一个区域，其中的颜色参数用来指定需要透明显示的颜色，然后调用frame窗口的SetShape函数设置这个区域模板。

```
void ShapedFrame::SetWindowShape()
{
    wxRegion region(m_bmp, *wxWHITE);
    m_hasShape = SetShape(region);
}
```

为了让这个窗口可以通过鼠标拖拽来实现在窗口上的移动，我们可以通过下面的鼠标移动事件处理函数：

```
void ShapedFrame::OnMouseMove(wxMouseEvent& evt)
{
    wxPoint pt = evt.GetPosition();
    if (evt.Dragging() && evt.LeftIsDown())
    {
        wxPoint pos = ClientToScreen(pt);
        Move(wxPoint(pos.x - m_delta.x, pos.y - m_delta.y));
    }
}
```

而重画事件处理函数就比较简单了，当然在你的真实的应用程序里，你可以在其中放置更多的图形元素。

```
void ShapedFrame::OnPaint(wxPaintEvent& evt)
{
    wxPaintDC dc(this);
    dc.DrawBitmap(m_bmp, 0, 0, true);
}
```

你也可以参考wxWidgets的发行版中的samples/shaped目录中的例子。

小型frame窗口

在Window平台和GTK+平台上，你可以使用wxMiniFrame来实现那些必须使用小的标题栏的窗口，例如，来实现一个调色板工具。下图(译者注：这个图片应该是搞错了，是下一个例子中的图片，不过作者的翻译源使用的就是这个图片，没有办法了。)演示了windows平台上的这种小窗口的样子。而在Max Os X平台上，这个小窗口则是直接使用的普通的frame窗口代替。其它方面wxMiniFrame和wxFrame都是一样的。

.tmp)

wxMDIParentFrame

这个窗口类继承自wxFrame,是wxWidgets的MDI（多文档界面）体系的组成部分。MDI的意思是由一个父窗口管理零个或多个子窗口的一种界面架构。这种MDI界面结构依平台的不同而有不同的外观，下图演示了两种主要的外观。在windows平台上，子文档窗口是排列在其父窗口内的一组frame窗口，这些文档窗口可以被平铺，层叠或者把其中的某一个在父窗口的范围内最大化以便在某个时刻仅显示一个文档窗口。 wxWidgets会自动在主菜单上增加一组菜单用来控制文档窗口的这些操作。MDI界面的一个优点是使得整个应用程序界面相对来说显得不那么零乱。这一方面是由于因为所有的文档窗口被限制在父窗口以内，另外一个方面，父窗口的菜单条会被活动的文档窗口的菜单条替代，这使得多个菜单条的杂乱性也有所减轻。

.tmp)

而在GTK+平台上，wxWidgets则通过TAB页面控件来模拟多文档界面。在某个时刻只能有一个窗口被显示，但是用户可以通过TAB页面在窗口之间进行切换。在Mac Os上，MDI的父窗口和文档窗口则都使用和普通的Frame窗口一样的外观，这符合这样的一个事实就是在Mac Os上，文档总是在一个新的窗口中被打开。

在那些MDI的文档窗口包含在父窗口之中的平台上，父窗口将把它的所有的文档窗口排列在一个子窗口中，而这个窗口可以和frame窗口中的其它控件子窗口和平共处。在上图中，父窗口将一个文本框控件和那些文档窗口进行了这样的排列。更详细的情形请参考wxWidgets发行版本的 samples/mdi目录中的例子。

除了父窗口的菜单条外，每一个文档窗口都可以有自己的菜单条。当某个文档窗口被激活时，它的菜单条将显示在父窗口上，而没有子文档窗口被激活时，父窗口显示自己的菜单条。在构建子文档窗口的菜单条时，你应该主要要包含那些同样命令的父窗口的菜单项，再加上文档窗口自己的命令菜单项。父窗口和文档窗口也可以拥有各自的工具条和状态栏，但是这两者并没有类似菜单条这样的机制。

wxMDIParentFrame类的构造函数和wxFrame类的构造函数是完全一样的.

wxMDIParentFrame的窗口类型

wxMDIParentFrame类额外的窗口类型列举如下:

| --- | --- | | wxFRAME_NO_WINDOW_MENU | Under Windows, removes the Window menu that is normally added automatically. |

wxMDIParentFrame的成员函数

下面列出了wxMDIParentFrame类除了wxFrame的函数以外的主要成员函数。

ActivateNext和ActivatePrevious函数激活前一个或者后一个子文档窗口。

Cascade和Tile层叠或者平铺所有的子窗口. ArrangeIcons函数以图标的方式平铺所有最小化的文档窗口.这三个函数都只适用于Windows平台.

GetActiveChild获取当前活动窗口的指针(如果有的话).

GetClientWindow函数返回那个包含所有文档窗口的子窗口的指针.这个窗口是自动创建的,但是你还是可以通过重载 OnCreateClient函数来返回一个你自己的继承自wxMDIClientWindow的类的实例。如果你这样作，你就需要使用两步创建的方法创建这个多文档父窗口类.

wxMDIChildFrame

wxMDIChildFrame窗口应该总被创建为一个wxMDIParentFrame类型窗口的子窗口. 正如我们前面已经提到的那样，依平台的不同，这种类型的窗口既可以是在其父窗口范围内的一组窗口列表，也有可能是在桌面上自由飞翔的普通的frame窗口。

除了父窗口必须不能为空，它的构造函数和普通的frame的构造函数是一样的。

```
#include "wx/mdi.h"
wxMDIParentFrame* parentFrame = new wxMDIParentFrame(
     NULL, ID_MYFRAME, wxT("Hello wxWidgets"));
wxMDIChildFrame* childFrame = new wxMDIChildFrame(
     parentFrame, ID_MYCHILD, wxT("Child 1"));
childFrame->Show(true);
parentFrame->Show(true);
```

wxMDIChildFrame的窗口类型

wxMDIChildFrame和wxFrame的窗口类型是一样的。尽管如此，不是所有的窗口类型的值在各个平台上都是有效的。

wxMDIChildFrame的成员函数

wxMDIChildFrame的除其基类wxFrame以外的主要成员函数列举如下:

Activate函数激活本窗口，将其带到前台并且使得其父窗口显示它的菜单条。

Maximize函数将其在父窗口范围内最大化(仅适用于windows平台).

Restore函数使其从最大化的状态恢复为普通状态(仅适用于windows平台)。

wxDialog

对话框是一种用来提供信息或者选项的顶层窗口。他可以有一个可选的标题栏，标题栏上可以有可选的关闭或者最小化按钮，和frame窗口一样，你也可以给它指定一个图标以便显示在任务栏或者其它的地方。一个对话框可以组合任何多个非顶层窗口或者控件，举例来说，可以包含一个wxNoteBook控件和位于底部的两个确定和取消按钮。正如它的名字所指示的那样，对话框的目的相比较于整个应用程序的主窗口，只是为了给用户提示一些信息，提供一些选项等。

有两种类型的对话框：模式的或者非模式的。一个模式的对话框在应用程序关闭自己之前阻止其它的窗口处理来自应用程序的消息。而一个非模式的对话框的行为则更像一个frame窗口。它不会阻止应用程序中的别的窗口继续处理消息和事件。要使用模式对话框，你应该调用ShowModal函数来显示对话框，否则就应该象frame窗口那样，使用Show函数来显示对话框。

模式对话框是wxWindow派生类中极少数允许在堆栈上创建的对象之一，换句话说，你既可以使用下面的方法使用模式对话框:

```
void AskUser()
{
    MyAskDialog *dlg = new MyAskDialog(...);
    if ( dlg->ShowModal() == wxID_OK )
        ...
    dlg->Destroy();
}
```

也可以直接使用下面的方法:

```
void AskUser()
{
    MyAskDialog dlg(...);
    if ( dlg.ShowModal() == wxID_OK )
        ...
    //这里不需要调用Destroy()函数
}
```

通常你应该从wxDialog类派生一个你自己的类，以便你可以更方便的处理类似wxEVT_CLOSE这样的事件以及其它命令类型的事件。通常你应该在派生类的构造函数中创建你的对话框需要的其它控件。

和wxFrame类一样，如果你的对话框只有一个子窗口，那么wxWidgets会自动为你布局这个窗口，但是如果有超过一个子窗口，应用程序应该自己负责窗口的布局（参考第7章）

当你调用Show函数时，wxWidgets会调用InitDialog函数来发送一个wxInitDialog事件到这个窗口，以便开始对话框数据传输和验证以及其它的事情。

除了默认的构造函数，wxDialog还拥有下面的构造函数:

```
wxDialog(wxWindow* parent, wxWindowID id, const wxString& title,
    const wxPoint& pos = wxDefaultPosition,
    const wxSize& size = wxDefaultSize,
    long style = wxDEFAULT_DIALOG_STYLE,
    const wxString& name = wxT("dialog"));
```

例如:

```
wxDialog* dialog = new wxDialog(NULL, ID_MYDIALOG,
    wxT("Hello wxWidgets"), wxDefaultPosition,
    wxSize(500, 300));
dialog->Show(true);
```

在调用Show(true)函数或者ShowModal函数之前，对话框都是不可见的，以便其可以以不可见的方式进行窗口布局。

默认情况下，如果对话框的父窗口为NULL,应用程序会自动指定其主窗口为其父窗口，你可以通过指定wxDIALOG_NO_PARENT类型来创建一个无父无母的孤儿dialog类，不过对于模式对话框来说，最好不要这样作。

和wxFrame类一样，最好不要直接使用delete操作符删除一个对话框，取而代之使用Destroy或者Close。以便在对话框的所有事件处理完毕以后才释放对话框，当你调用Cloes函数时，默认发送的wxEVT_CLOSE事件的处理函数其实也是调用了Destroy函数.

需要注意，当一个模式的对话框被释放的时候，它的某个事件处理函数需要调用EndModal函数，这个函数的参数是导致这次释放的那个事件所属窗口的窗口标识符（例如wxID_OK或者wxID_CANCEL).这样才能使得应用程序退出这个模式对话框的事件处理循环，从而使得调用ShowModal的代码能够释放这个窗口。而这个标识符则作为ShowModal函数的返回值。我们举下面一个OnCancel函数为例：

```
//wxID_CANCEL的命令事件处理函数
void MyDialog::OnCancel(wxCommandEvent& event)
{
    EndModal(wxID_CANCEL);
}
//显示一个模式对话框
void ShowDialog()
{
    //创建这个对话框
    MyDialog dialog;
    // OnCancel函数将会在ShowModal执行的过程中被调用。
    if (dialog.ShowModal() == wxID_CANCEL)
    {
        ...
    }
}
```

下图展示了几个典型的对话框，我们会在第9章阐述它们的创建过程。当然，对话框可以远比图中的那些更复杂，例如下面第二个图中的那样，它包含一个分割窗口，一个树形控件以便显示不同的面板(panels)，以及一个扮演属性编辑器的网格控件

.tmp)

.tmp)

wxDialog的窗口类型

除了基本窗口类型以外，wxDialog还有下面一些窗口类型可以使用：

wxDEFAULT_DIALOG_STYLE	其值等于**wxSYSTEM_MENU | wxCAPTION | wxCLOSE_BOX.**
wxCAPTION	在对话框上显示标题.
wxMINIMIZE_BOX	在标题栏显示最小化按钮.
wxMAXIMIZE_BOX	在标题栏显示最大化按钮.
wxCLOSE_BOX	在标题栏显示关闭按钮.
wxSTAY_ON_TOP	对呼框总在最前. 仅支持windows平台.
wxSYSTEM_MENU	显示系统菜单.
wxRESIZE_BORDER	显示可变大小边框.
wxDIALOG_NO_PARENT	如果创建对话框的时候父窗口为NULL，则应用程序会使用其主窗口作为对话框的父窗口，使用这个类型可以使得在这种情况下，对话框强制使用NULL作为其父窗口。模式窗口不推荐强制使用NULL作为父窗口。

下表解释了对话框类额外的扩展窗口类型。虽然wxWS_EX_BLOCK_EVENTS类型是窗口基类的窗口类型，但是由于它是对话框类的默认扩展类型，我们在这里也进行了描述。

wxDIALOG_EX_CONTEXTHELP	在**Windows**平台上**,**这个扩展类型使得对话框增加一个查询按钮**.**如果这个按钮被按下，对话框将进入一种帮助模式，在这种模式下，无论用户点击哪个子窗口，都将发送一个**wxEVT_HELP**事件**.** 这个扩展类型不可以和**wxMAXIMIZE_BOX**和**wxMINIMIZE_BOX**类型混用**.**
wxWS_EX_BLOCK_EVENTS	这是一个默认被设置的扩展类型。意思是阻止命令类型事件向更上一级的窗口传播.要注意调用SetExtraStyle函数可以重置这个扩展类型.
wxDIALOG_EX_METAL	在Mac OS X平台上,这个扩展类型导致对话框使用金属外观.不要滥用这个类型，因为它默认用户的机器有各种多媒体硬件

wxDialog事件

下表解释了相对于窗口基类来说额外的对话框事件:

EVT_ACTIVATE(func)	用于处理**wxEVT_ACTIVATE**事件,在对话框即将激活或者去激活的时候产生。处理函数的参数类型是**wxActivateEvent**。
EVT_CLOSE(func)	用户处理wxEVT_CLOSE事件,在应用程序或者操作系统即将关闭窗口的时候产生.处理函数的参数类型是wxCloseEvent， 这个事件可以调用Veto函数来放弃事件的后续处理.
EVT_ICONIZE(func)	用来处理wxEVT_ICONIZE事件, 在窗口被最小化或者从最小化状态恢复的时候产生.处理函数的参数类型是wxIconizeEvent. 调用IsIconized来检测到底是最小化还是从最小化恢复.
EVT_MAXIMIZE(func)	用来处理wxEVT_MAXIMIZE事件,这个事件在窗口被最大化或者从最大化状态恢复的时候产生， 处理函数的参数类型是wxMaximizeEvent.调用IsMaximized函数确定具体的状态类型.
EVT_INIT_DIALOG(func)	用于处理wxEVT_INIT_DIALOG事件,在对话框初始化自己之前被产生。处理函数的参数类型是wxInitDialogEvent.wxPanel也会产生这个事件.默认执行的操作是调用TransferDataToWindow.

4.5 容器窗口

容器窗口是用来装载别的可见元素的窗口，所谓可见元素指的是子窗口或者是画在这个窗口上的图案。

wxPanel

wxPanel是一个在某些方面有点象dialog窗口的窗口。这个窗口通常被用来摆放那些除了对话框或者frame窗口以外的其它控件窗口。它也常被用来作为wxNoteBook控件的页面。它通常使用系统默认的颜色。

和对话框一样，可以使用它的InitDialog方法来产生一个wxInitDialogEvent事件。如果设置了wxTAB_TRAVERSAL类型，那么它通常可以通过使用类似TAB键的导航键遍历所有它上面的子控件。

除了默认的构造函数以外，wxPanel还拥有下面的构造函数：

```
wxPanel(wxWindow* parent, wxWindowID id,
    const wxPoint& pos = wxDefaultPosition,
    const wxSize& size = wxDefaultSize,
    long style = wxTAB_TRAVERSAL|wxNO_BORDER,
    const wxString& name = wxT("panel"));
```

用法如下:

```
wxPanel* panel = new wxPanel(frame, wxID_ANY,
    wxDefaultPosition, (500, 300));
```

wxPanel的窗口类型

wxPanel没有额外的窗口类型。

wxPanel的成员函数

wxPanel也没有额外的成员函数。

wxNotebook

这个类提供了一个有多个页面的窗口，页面之间可以通过边上的TAB按钮来切换。每个页面通常是一个普通的wxPanel窗口或者其派生类，当然你完全可以使用别的窗口。

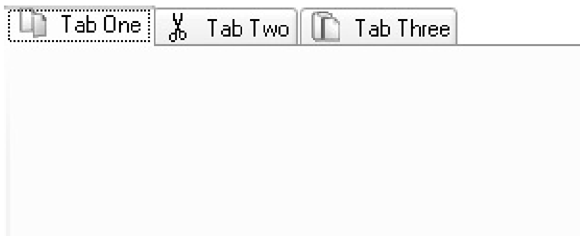
NoteBook的TAB按钮可以包含一个图片，也可以包含一个文本标签。图片是由wxImageList（参考第10章）提供的，是通过在列表中的位置和页面对应的。

使用notebook的方法是，创建一个wxNotebook对象，然后调用其AddPage方法或者InserPage方法，传递一个用来作为页面的窗口指针。不要手动释放那些已经被wxNoteBook作为页面的窗口，你应该使用DeletePage来删除某个页面或者干脆等到notebook释放它自己的时候。notebook会一并释放那些页面.

下面举例说明怎样创建一个有三个页面，包含文本和图片的TAB标签的notebook：

```
#include "wx/notebook.h"
#include "copy.xpm"
#include "cut.xpm"
#include "paste.xpm"
//创建notebook
wxNotebook* notebook = new wxNotebook(parent, wxID_ANY,
  wxDefaultPosition, wxSize(300, 200));
// 创建图片列表
wxImageList* imageList = new wxImageList(16, 16, true, 3);
imageList->Add(wxIcon(copy_xpm));
imageList->Add(wxIcon(paste_xpm));
imageList->Add(wxIcon(cut_xpm));
// 创建页面
wxPanel1* window1 = new wxPanel(notebook, wxID_ANY);
wxPanel2* window2 = new wxPanel(notebook, wxID_ANY);
wxPanel3* window3 = new wxPanel(notebook, wxID_ANY);
notebook->AddPage(window1, wxT("Tab one"), true, 0);
notebook->AddPage(window2, wxT("Tab two"), false, 1);
notebook->AddPage(window3, wxT("Tab three"), false 2);
```

下图演示了上述代码在windows平台上的结果：



在大多数的平台上，当TAB页面数量太多不能被完全显示的时候，都会自动出现导航按钮。但是在Mac OS上，这个导航按钮是不会出现的，因此在这个系统上，可以使用的页面数目将受到窗口大小和TAB标签大小的限制。

如果你在每个页面上都使用布局控件，并且在创建notebook的时候使用默认的大小wxDefaultSize,那么wxNotebook将会自动调整大小以适合它的页面。

Notebook窗口主题管理

在Windows Xp系统中，默认的窗口主题会给notebook的页面添加过渡色。虽然这是预期的本地行为，但是它可能会降低性能。出于审美方面的原因，你可能想直接使用单一的色彩，尤其是当notebook不是位于一个对话框以内的时候。如果你想阻止这种默认的行为，可以采用以下三种办法：第一：你可以给你的 notebook指定wxNB_NOPAGETHEM类型来禁止某个特定的notebook的这种效果，或者你可以使用 wxSystemOptions::SetOption来在应用程序范围内禁止它，或者你可以使用SetBackgroundColour函数来给某个单独的页面禁止它。要在应用

程序范围内禁止它，你可以使用下面的代码：
wxSystemOptions::SetOption(wxT("msw.notebook.themed-background"), 0); 将它的值设置成1可以再次允许这种效果。要让某个单独的页面禁用这种效果可以使用下面的方法：

```
wxColour col = notebook->GetThemeBackgroundColour();
if (col.Ok())
{
    page->SetBackgroundColour(col);
}
```

在windows系统以外的平台，或者如果一个应用程序没有使用主题风格，GetThemeBackgroundColour都将返回一个未初始化的颜色，因此上面的代码在各个平台都是可以使用的。另外就是上述的这部分代码的语法和行为是有可能在将来的wxWidgets版本中发生变化，请参考你本地wxWidgets发行版中的wxNotebook类的文档来获取最新的信息。

wxNotebook的窗口类型

wxNotebook可以拥有下面的额外窗口类型:

wxNB_TOP	标签放在顶部.
wxNB_LEFT	标签放在左面. 不是所有的WindowsXp的主题都支持这个选项.
wxNB_RIGHT	标签在右面. 不是所有的WindowsXp的主题都支持这个选项.
wxNB_BOTTOM	标签在底部. 不是所有的WindowsXp的主题都支持这个选项.
wxNB_FIXEDWIDTH	所有标签宽度相同.仅适用于windows.
wxNB_MULTILINE	可以有多行的标签. 仅适用于windows
wxNB_NOPAGETHEME	在windos上禁用主题风格.以提高性能和提供另外一种审美选择.

wxNotebook的事件

wxNotebook可以产生wxNotebookEvent事件，这个事件可以被它和它的继承者处理.

EVT_NOTEBOOK_PAGE_CHANGED(id,func)	当前页面已经改变.
EVT_NOTEBOOK_PAGE_CHANGING(id,func)	当前页面即将改变。你可以使用Veto函数阻止这种改变.

wxNotebook的成员函数

AddPage增加一个页面,InsertPage在某个固定位置插入一个页面.你可以使用文本标签或者图片标签或者两者都有，用法如下:

```
//增加一个使用了文本和图片两种标签都有的页面，并且当前处于未选中状态。
// (其中图片使用的是图片列表中的索引为2的那个).
notebook->AddPage(page, wxT("My tab"), false, 2);
```

DeletePage函数移除并且释放某个特定的页面，而RemovePage函数则仅仅是移除这个页面。DeleteAllPages函数用来删除所有的页面。当wxNoteBook被释放时，它也会自动删除所有的页面。

AdvanceSelection函数循环选择页面。 SetSelection函数以基于0的索引选择特定的页面. GetSelection取得当前选中页面的索引或者返回wxNOT_FOUND.

SetImageList函数用来给Notebook设置一个图标列表，这个函数仅是设置而不绑定图片列表，这意味着在notebook 控件被删除的时候，这个图片列表控件并不会被删除，如果你希望使用绑定，则可以使用AssignImageList函数. GetImageList函数返回notebook相关的图片列表对象.图片列表对象用来提供每个页面标签中使用的图片，详情请参考第10章。

GetPage函数用来返回和某个索引对应的页面窗口指针, GetPageCount函数则返回页面总数.

SetPageText和GetPageText用来操作页面标签上的文本.

SetPageImage和GetPageImage用来操作页面标签上的图片在图片列表中的索引,

wxNotebook的替代选择

wxNotebook是wxBookCtrlBase的派生类, 这个基类是用来提供用于管理一组页面的所有数据和方法的抽象类。和wxNotebook有着相似API的类还有两个，一个是wxListbook，一个是wxChoicebook,你也可以实现你自己的类，比如你可以实现一个wxTreebook.

wxListbook使用一个wxListCtrl变量来控制页面;ListCtrl控件是一种在内部显示一组带有标签的图片的控件. 在wxListbook中，相关的ListCtrl控件可以显示在上下左右四个方向，默认是在左边。这是wxNotebook的一个很有吸引力的替代者。因为即使在Mac Os X平台上，ListCtrl可以管理的项目数量也几乎没有限制，因此wxListbook可以管理的页面数也不受窗口大小和标签长度的限制。

wxChoicebook则使用一个选择控件（一个下拉列表）来管理页面。比较适用于窗口空间比较小的场合。这个控件不会在页面标签处显示图片，而且默认情况下，选择控件显示在整个控件的上方。

上述这两个控件的头文件分别为wx/listbook.h和wx/choicebk.h.它们的事件处理函数的参数类型分别为 wxListbookEvent和wxChoicebookEvent类型，事件影射宏则分别为EVT_XXX_PAGE_CHANGED(id, func)和EVT_XXX_PAGE_CHANGING(id,func),其中XXX代表LISTBOOK或者CHOICEBOOK.

你可以使用类似wxNotebook定义的那些窗口类型，也可以类似用wxCHB_TOP或者wxLB_TOP取代wxNB_TOP这样的窗口类型，它们的值都是一样的,

wxScrolledWindow

尽管所有的窗口都可以拥有滚动条，但是为了让滚动条工作，还需要一些额外的代码。这个类则为让不同类型的窗口类中的滚动条正确工作提供了足够的灵活性。 wxScrolledWindow主要实现了那些让窗口可以以一定的单位连续的滚动（而不是不定大小的跳跃），也可以定义在使用翻页键进行翻页的时候的大小。这种实现通常仅适用于类似画图程序中的那种翻页功能，对于一些复杂功能的文本编辑程序则不一定适合。因为富文本编辑器中每一行的高度和宽度都有可能不同。wxGrid网格控件就是这样的一个例子（在网格控件中，每一行和每一列的宽度都有可能不同），在这种情况下，你应该自己直接从wxWindow类实现自己的派生类，从而实现自己的窗口滚动机制。

要适用滚动窗口，你需要提供每次逻辑移动的单位（也就是当处理上滚一行和下滚一行时窗口应该移动的大小）以及整个窗口的虚拟逻辑大小。然后wxScrollWindow类就会自己关注是否显示滚动条，以及滚动条上的滑钮应该用多大显示等等这些细节问题.

下面演示了怎样创建一个滚动窗口:

```
#include "wx/scrolwin.h"
wxScrolledWindow* scrolledWindow = new wxScrolledWindow(
    this, wxID_ANY, wxPoint(0, 0), wxSize(400, 400),
    wxVSCROLL|wxHSCROLL);
// 设置窗口的虚拟逻辑大小： 1000x1000
// 每次滚动10个象素
int pixelsPerUnixX = 10;
int pixelsPerUnixY = 10;
int noUnitsX = 1000;
int noUnitsY = 1000;
scrolledWindow->SetScrollbars(pixelsPerUnitX, pixelsPerUnitY,
    noUnitsX, noUnitsY);
```

第二种设置虚拟大小的方法是使用SetVirtualSize函数，它的参数是以象素为单位的虚拟大小。然后再用SetScrollRate 函数来设置水平和垂直方向上的滚动增量。第三种方法是使用布局控件来布局窗口，滚动窗口会自动计算所有子窗口需要的窗口大小作为窗口的虚大小，你同样需要调用SetScrollRate函数来设置滚动增量。

你可以想普通窗口那样使用重画事件，但是在进行任何窗口重画动作之前你应该调用DoPrepareDC函数来保证重画动作以当前的窗口原点作为原点，就象下面演示的那样：

```
void MyScrolledWindow::OnPaint(wxPaintEvent& event)
{
    wxPaintDC dc(this);
    DoPrepareDC(dc);
    dc.SetPen(*wxBLACK_PEN);
    dc.DrawLine(0, 0, 100, 100);
}
```

你也可以直接重载OnDraw虚函数，wxScrolledWindow在调用这个函数之前，会首先调用DoPrepareDC函数，因此你只需要象下面演示的那样作：

```
void MyScrolledWindow::OnDraw(wxDC& dc)
{
    dc.SetPen(*wxBLACK_PEN);
    dc.DrawLine(0, 0, 100, 100);
}
```

需要注意的是，在别的任何事件处理函数中如果要重画窗口，你同样需要调用DoPrepareDC函数。

你也可以象下面这样提供你自己的DoPrepareDC函数，这个函数默认的行为只是把设备操作原点移动到当前滚动条开始的位置：

```
void wxScrolledWindow::DoPrepareDC(wxDC& dc)
{
    int ppuX, ppuY, startX, startY;
    GetScrollPixelsPerUnit(& ppuX, & ppuY);
    GetViewStart(& startX, & startY);
    dc.SetDeviceOrigin( - startX * ppuX, - startY * ppuY );
}
```

关于在wxScrollWindow上进行作图的详细情形，包括怎样使用双缓冲区，请参考第5章，ï¿½重画和打印ï¿½中的wxPaintDC小节。

wxScrolledWindow的窗口类型

wxScrolledWindow类并没有特别的窗口类型，但是通常需要设置wxVSCROLL|wxHSCROLL类型，这也是wxScrolledWindow的默认类型。在某些平台上因为效率的原因可能不支持这两个类型。

wxScrolledWindow的事件

wxScrolledWindow会产生wxScrollWinEvent事件（参见下表）.这些事件不会在父子窗口关系中传播，因此要处理这种事件，你必须定义自己的滚动窗口派生类或者挂载你自己的事件表。不过在通常情况下，你并不需要覆盖默认的处理函数来自己处理这些事件。

EVT_SCROLLWIN(func)	处理所有滚动事件.
EVT_SCROLLWIN_TOP(func)	处理事件wxEVT_SCROLLWIN_TOP，滚动到最顶端.
EVT_SCROLLWIN_BOTTOM(func)	处理事件wxEVT_SCROLLWIN_BOTTOM 滚动到最底端事件.
EVT_SCROLLWIN_LINEUP(func)	处理事件wxEVT_SCROLLWIN_LINEUP上滚一行.
EVT_SCROLLWIN_LINEDOWN(func)	处理事件wxEVT_SCROLLWIN_LINEDOWN 下滚一行.
EVT_SCROLLWIN_PAGEUP(func)	处理事件wxEVT_SCROLLWIN_PAGEUP 上滚一页.
EVT_SCROLLWIN_PAGEDOWN(func)	处理事件 wxEVT_SCROLLWIN_PAGEDOWN 下滚一页.

wxScrolledWindow的成员函数介绍

CalcScrolledPosition和CalcUnscrolledPosition函数都需要四个参数，前两个整数参数是输入需要计算的点，后两个则是指向整数的指针用来放置计算结果。第一个函数用来计算实际位置到逻辑位置的映射。如果当前的滚动条下滚了10个象素，则0这个数字作为输入将得到对应输出-10,第二个函数则实现相反的功能。

EnableScrolling函数用来允许或者禁止垂直或者水平方向的物理滚动。物理滚动的含义是说在收到滚动事件的时候对窗口进行物理上的平移。但是如果应用程序需要不等量的移动（比如，由于字体的不同为了避免滚动的时候显示半行字的情况出现），则需要禁止物理移动，在这种情况下，应用程序需要自己移动对应的子窗口。物理移动在所有支持物理移动的平台上都是默认打开的。当然不是所有的平台都支持物理移动。

GetScrollPixelsPerUnit函数在两个指向整数的指针中返回水平和垂直方向上的移动单位。返回0表示在那个方向上不能滚动。

GetViewStart函数返回窗口可视部分左上角的座标，单位是逻辑单位，你需要乘以GetScrollPixelsPerUnit的返回值以便将结果转换成象素单位。

GetVirtualSize返回当前设定的以象素为单位的虚拟窗口大小。

DoPrepareDC将画画设备的原点设置到当前的可见原点。

Scroll函数将窗口滚动到一个特定的逻辑单位位置（注意这里的单位不是象素）.

SetScrollbars用来设置移动单位的象素值,各个方向的移动单位总数,水平或者垂直方向的当前滚动位置(可选)以及是否立即刷新窗口（默认否）等。

SetScrollRate用来设置滚动单位，相当于单独设置SetScrollbars中的移动单位参数.

SetTargetWindow用来滚动非wxScrolledWindow类型的其它窗口.

滚动非wxScrolledWindow类型的窗口

如果你想自己实现窗口的滚动行为，你可以直接从wxWindow派生你的窗口类，然后使用SetScrollbar函数来设置这个窗口的滚动条。

SetScrollbar函数的参数如下表的说明：

int orientation	滚动条的类型**: wxVERTICAL or wxHORIZONTAL.**
int position	滚动条滑块的位置，逻辑滚动单位.
int visible	可见部分大小，逻辑滚动单位. 通常会决定滑块的长度.
int range	滚动条的最大长度，逻辑滚动单位.
bool refresh	是否立即刷新窗口.

举例来说，如果你想显示一个50行文本的文本窗口，使用同样的字体，而窗口的大小只够显示16行，你可以使用下面的代码：

```
SetScrollbar(wxVERTICAL, 0, 16, 50)
```

注意在上面的例子中，滑块的位置永远不可能大于50-16=34.

你可以通过用当前窗口除以当前字体下文本的高度来得到当前窗口可以显示多少行这个值。

如果你是自己实现滚动行为，你总是需要在窗口大小发生改变的时候更改滚动条的设置。因此你可以在首次计算滚动条参数的代码中使用AdjustScrollbars函数，然后在wxSizeEvent的处理事件中使用AdjustScrollbars函数。

你可以参考wxGrid控件的代码来获得实现自定义滚动窗口的更多灵感。

你也可以参考wxWidgets手册中的wxVScrolledWindow类，它可以用来建立一个可以在垂直方向进行不固定单位滚动的窗口类。

wxSplitterWindow

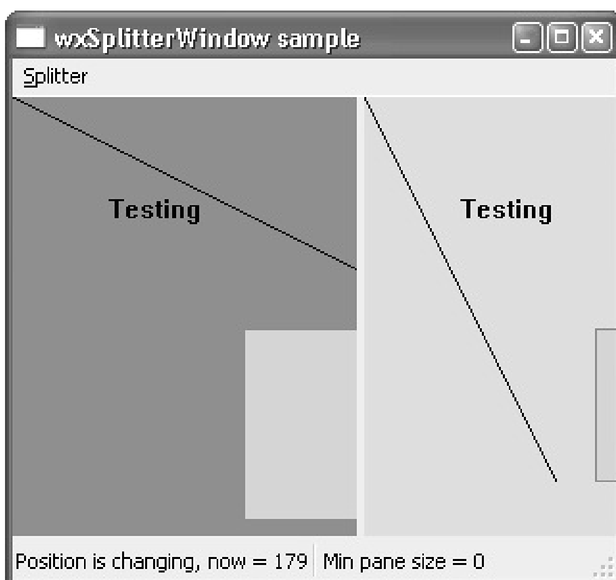
这个类用来管理最多两个窗口，如果你想在更多窗口中实现分割，你可以使用多个分割窗口。当前的窗口可以被应用程序分割成两个窗口，比如通过一个菜单命令，也可以通过应用程序命令或者通过分割窗口的用户操作界面（双击分割条或者拖拽分割条使得其中一个窗口的大小变为0）重新变为一个窗口.其中拖拽分割条的方法将会受到后面会提到的SetMinimumPaneSize方法的限制。

在大多数平台上，当分割条被拖拽时，会有一个和背景颜色相反的竖条随之移动以显示分割条的最终位置，你可以通过使用 wxSP_LIVE_UPDATE窗口类型来使得分割条以实时方式通过直接改变两个窗口的大小来代替那种默认方式。实时方式是Mac OS X上默认的也是唯一的方式。

下面的代码演示了怎样创建一个分割窗口来操作两个窗口并且隐藏其中的一个：

```
#include "wx/splitter.h"
wxSplitterWindow* splitter = new wxSplitterWindow(this, wxID_ANY,
    wxPoint(0, 0), wxSize(400, 400), wxSP_3D);
leftWindow = new MyWindow(splitter);
leftWindow->SetScrollbars(20, 20, 50, 50);
rightWindow = new MyWindow(splitter);
rightWindow->SetScrollbars(20, 20, 50, 50);
rightWindow->Show(false);
splitter->Initialize(leftWindow);
//去掉下面的注释以便禁止窗口隐藏
//    splitter->SetMinimumPaneSize(20);
下面的代码代码演示了创建分割窗口以后怎样使用它：
void MyFrame::OnSplitVertical(wxCommandEvent& event)
{
    if ( splitter->IsSplit() )
        splitter->Unsplit();
    leftWindow->Show(true);
    rightWindow->Show(true);
    splitter->SplitVertically( leftWindow, rightWindow );
}
void MyFrame::OnSplitHorizontal(wxCommandEvent& event)
{
    if ( splitter->IsSplit() )
        splitter->Unsplit();
    leftWindow->Show(true);
    rightWindow->Show(true);
    splitter->SplitHorizontally( leftWindow, rightWindow );
}
void MyFrame::OnUnsplit(wxCommandEvent& event)
{
    if ( splitter->IsSplit() )
        splitter->Unsplit();
}
```

下图演示了分割窗口在windows平台上的例子，在这个例子中，分割窗口没有使用wxSP_NO_XP_THEME类型。如果使用了这个类型，分割窗口将会拥有更传统的下沉边框和3维外观。



wxSplitterWindow的窗口类型

wxSP_3D	使用三维效果的边框和分割条 .
wxSP_3DSASH	使用三维效果分割条 .
wxSP_3DBORDER	和xSP_BORDER效果相同
wxSP_BORDER	使用标准边框 .
wxSP_NOBORDER	无边框(默认值).
wxSP_NO_XP_THEME	在Xp操作系统上，如果你不喜欢默认的效果，使用三维边框和分割条效果。
wxSP_PERMIT_UNSPLIT	即使设置了最小值也允许窗口被隐藏.
wxSP_LIVE_UPDATE	在分割条移动的时候实时更新窗口.

wxSplitterWindow事件

wxSplitterWindow使用wxSplitterEvent类型的事件处理函数

EVT_SPLITTER_SASH_POS_CHANGING(id,func)	用于处理 **wxEVT*COMMAND_SPLITTER_S* POS_CHANGING**事件,在分割条位 改变的时候产生。调用**Veto**函数阻 移动，或者调用事件的**SetSashPo** 数来更改分割条的位置.
EVT_SPLITTER_SASH(id,func)	用于处理wxEVT*COMMAND_SPLI* SASH_POS_CHANGED事件,分割 置已经改变你可以通过事件的 SetSashPosition函数阻止这种改变 改变化的幅度.
EVT_SPLITTER_UNSPLIT(id,func)	用于处理 wxEVT_COMMAND_SPLITTER_U 事件,在有某个窗口被隐藏的时候产

EVT_SPLITTER_DCLICK(id,func) | 用于处理 wxEVT_COMMAND_SPLITTER_DOUBLECLICKED事件,在分割条被双击的时候产生. |

wxSplitterWindow的成员函数

GetMinimumPaneSize和SetMinimumPaneSize函数用于操作分割窗口的最小窗格大小。默认为0,意味着分割窗口的每一侧的大小都可以被缩减至0。相当于移除某一部分分割窗口.要阻止这种情形，也为了阻止拖拽分割条的时候超出边界，可以将其设置成一个大于0的值比如20个象素。虽然这样，如果设置了wxSP_PERMIT_UNSPLIT窗口类型，即使最小值设置为大于0的数，也还是可以将某个分割窗口隐藏。

GetSashPosition和SetSashPosition用来操作分割条的位置，传递true参数给SetSashPosition函数导致分割条被立刻刷新。

GetSplitMode和SetSplitMode函数用来设置分割的方向wxSPLIT_VERTICAL或者wxSPLIT_HORIZONTAL.

GetWindow1和GetWindow2用来获取两个分割窗口的指针。

Initialize函数使用一个窗口指针参数调用，用来显示两个分割窗口中的某一个窗口。

IsSplit函数用来判断窗口是否处于分割状态.

ReplaceWindow用来替换被分割窗口控制的两个窗口中的一个。使用这个函数要好过先使用Unsplit函数然后再增加另外一个窗口.

SetSashGravity用一个浮点小数来设置分割比例。0.0表示只显示右边或者下边的窗口，1.0表示只显示左边或者上边的窗口。中间的值则表示按照某种比例分配两个窗口的大小。使用GetSashGravity函数来获取当前的分割比例。

SplitHorizontally函数SplitVertically使用一个可选的分割尺寸来初始化分割窗口。

Unsplit去掉指定的那个分割窗口.

UpdateSize用来使得分割窗口立即刷新(通常情况下，这是在系统空闲的时候完成的).

布局控件中使用wxSplitterWindow的说明

在布局控件中使用分割窗口有一点细微之处需要说明一下。如果你不需要这个分割窗口的分割条是可移动的，你可以在创建两个子窗口的时候指定绝对大小。这将固定这两个子窗口的最小大小，从而使得分割条不能自由移动。如果你希望分割条能正常移动，在创建两个子窗口的时候你就需要使用默认的大小，然后在分割窗口的构造函数中指定其最小大小。然后在将这个分割窗口增加到布局控件的时候，在Add函数中使用wxFIXED_MINSIZE标记来告诉wxWidgets 将分割窗口控件当前的大小作为其最小大小。

另外一种情况是，分割窗口没有设定分割条的位置，也没有设定它的子窗口的大小，当布局控件完成布局时，分割窗口不愿意过早的设置分割条的位置，而要等到系统空闲的时候（这是它的默认行为）。在这种情况下，我们可能会看到当窗口刚刚可见的时候分割条复位自己的位置。为了避免出现这种情景，我们应该在对布局控件调用Fit之后，马上调用分割窗口的UpdateSize函数让其作立即更新。

默认情况下，当用户或者应用程序改变分割窗口的大小时，只有底端（或者右端）的子窗口改变自己的大小以便获取或减小额外的空间，要改变这种默认的行为，你可以使用前面说过的SetSashGravity函数指定一个固定的分割比例。

wxSplitterWindow的替代者

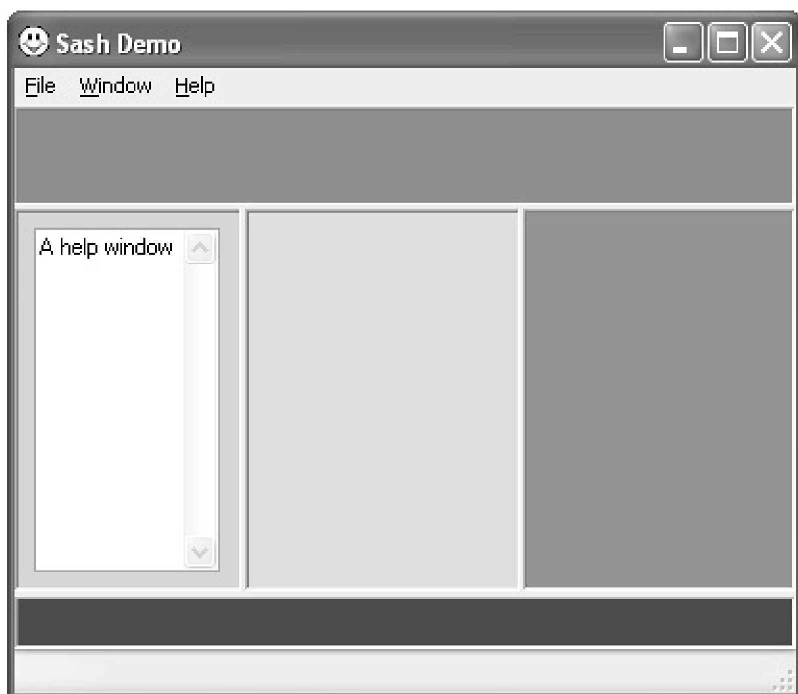
如果在应用程序中你需要使用很多的分割窗口，不妨考虑使用wxSashWindow.这个窗口允许它的任何边成为一个分割条，以便将其分割成多个窗口。而新分的这些窗口通常是用户创建的wxSashWindow的子窗口。

当wxSashWindow的分割条被拖动时，会向应用程序发送wxSashEvent事件以便事件处理函数可以相应的进行窗口布局。布局是通过一个称为wxLayoutAlgorithm的类来完成的，这个类可以根据父窗口的不同提供LayoutWindow，LayoutFrame， LayoutMDIFrame等多种不同的排列方法。

你还可以使用wxSashLayoutWindow类，这个类通过wxQueryLayoutInfoEvent事件来给wxLayoutAlgorithm提供布局方向和尺寸方面的信息。

关于wxSashWindow，wxLayoutAlgorithm和wxSashLayoutWindow更多的信息请参考手册中的相关内容。wxSashWindow不允许子窗口被移动或者分离，因此在不久的将来，这个类可能被一个通用的支持合并和分离的布局框架体系所代替。

下图演示了samples/sashtest目录中的例子在windows上执行的效果：



4.6 非静态控件

非静态控件指的是那些可以响应鼠标和键盘事件的类似wxButton和wxListBox之类的控件。这里我们会对其中最基本的那些进行一些描述。更高级的内容被安排在第12章，你可以参考附录E中的方法从网上下载或者创建你自己的更高级的控件。

wxButton

wxButton控件看上去象是一个物理上可以按下的按钮，它拥有一个文本标签，是用户界面中最常用的一个元素，它可以被放置在对话框，或者面板(wxPanel)或者几乎任何其它的窗口中。当用户点击按钮的时候，会产生一个命令类型的事件。

下面是创建按钮的一个简单的例子:

```
#include "wx/button.h"
wxButton* button = new wxButton(panel, wxID_OK, wxT("OK"),
    wxPoint(10, 10), wxDefaultSize);
```

下图演示了按钮在Windows Xp系统上的默认外观：



wxWidgets创建的按钮的默认大小是依靠静态函数wxButton::GetDefaultSize指定的，这个函数在不同的平台上返回不同的值。不过你可以通过给按钮指定wxBU_EXACTFIT类型来使其产生刚好符合其标签尺寸的大小。

wxButton的窗口类型

wxBU_LEFT	标签文本作对齐. 仅适用于**Windows**和**GTK+**平台.
wxBU_TOP	标签文本上对齐. 仅适用于Windows和GTK+平台.
wxBU_RIGHT	标签文本右对齐。仅适用于Windows和GTK+.
wxBU_BOTTOM	标签文本对齐按钮底部. 仅适用于Windows和GTK+平台.
wxBU_EXACTFIT	不使用默认大小创建按钮，而是按照其标签文本的大小来创建按钮.
wxNO_BORDER	创建一个平面按钮。仅适用于Windows和GTK+平台.

wxButton的事件

wxButton可以创建类型为wxCommandEvent的事件，如下表所示：

EVT_BUTTON(id,func)	用于处理wxEVT_COMMAND_BUTTON_CLICKED事件,在用户用左键单击按钮的时候产生.

wxButton的成员函数

wxButton的成员函数

SetLabel和GetLabel函数用来操作按钮标签文本.你可以使用"&"前导符来指示用于这个按钮的快捷键，仅适用于Windows和GTK+平台. SetDefault将按钮设置为其父窗口的默认按钮，这样用户按回车键就相当于用左键点击了这个按钮.

wxButton的标签

你可以使用一个前导符"&"来指定其后的字符为这个按钮的快捷键，不过这个操作仅适用于Windows和GTK+的版本，在其它的平台上，这个前导符将被简单的忽略。

在某些系统，尤其是GTK+系统中，例如确定或者新建这样的标准按钮的标签以及附带显示的一些小图片都被进行了默认的定义。在wxWidgets中，通常 你只需要指定这个按钮的标识符为系统预定义的标识符，那么这些预定义的标签和小图片将被显示，不过，指定和默认值不同的标签文本以及小图片的操作都是允许 的。

推荐的对于这些预定义按钮的使用方法如下，你只需要指定预定义的标识符，不需要指定标签文本，或者指定标签文本为空字符串：

```
wxButton* button = new wxButton(this, wxID_OK);
```

wxWidgets会各种平台提供合适的标签，在上面的例子中，在windows或者Mac OSX平台上，将使用字符串"&OK",而在GTK+系统中，会使用当前语言下的OK按钮标签。然后如果你提供了自己的标签文本， wxWidgets将会按照你的指示办事：

```
wxButton* button = new wxButton(this, wxID_OK, wxT("&Apply"));
```

这会导致无论在哪种平台上，这个按钮都将覆盖标准标识符的定义，使用"Apply"作为这个按钮的标签。

你可以使用wxGetStockLabel函数(需要包含wx/stockitem.h这个头文件)来获得某个预定义窗口标识符的标签，使用窗口标识符和true（如果你希望在返回的结果中包含前导符(&)）作为参数.

下表列出了预定义标识符和其默认标签的对应关系。

预定义标识符	预定义标签
wxID_ADD	"Add"
wxID_APPLY	"&Apply"
wxID_BOLD	"&Bold"
wxID_CANCEL	"&Cancel"

wxID_CLEAR	"&Clear"
wxID_CLOSE	"&Close"
wxID_COPY	"&Copy"
wxID_CUT	"Cu&t"
wxID_DELETE	"&Delete"
wxID_FIND	"&Find"
wxID_REPLACE	"Rep&lace"
wxID_BACKWARD	"&Back"
wxID_DOWN	"&Down"
wxID_FORWARD	"&Forward"
wxID_UP	"&Up"
wxID_HELP	"&Help"
wxID_HOME	"&Home"
wxID_INDENT	"Indent"
wxID_INDEX	"&Index"
wxID_ITALIC	"&Italic"
wxID_JUSTIFY_CENTER	"Centered"
wxID_JUSTIFY_FILL	"Justified"
wxID_JUSTIFY_LEFT	"Align Left"
wxID_JUSTIFY_RIGHT	"Align Right"
wxID_NEW	"&New"
wxID_NO	"&No"
wxID_OK	"&OK"
wxID_OPEN	"&Open"
wxID_PASTE	"&Paste"
wxID_PREFERENCES	"&Preferences"
wxID_PRINT	"&Print"
wxID_PREVIEW	"Print previe&w"
wxID_PROPERTIES	"&Properties"
wxID_EXIT	"&Quit"
wxID_REDO	"&Redo"
wxID_REFRESH	"Refresh"

wxID_REMOVE	"Remove"
wxID_REVERT_TO_SAVED	"Revert to Saved"
wxID_SAVE	"&Save"
wxID_SAVEAS	"Save &As..."
wxID_STOP	"&Stop"
wxID_UNDELETE	"Undelete"
wxID_UNDERLINE	"&Underline"
wxID_UNDO	"&Undo"
wxID_UNINDENT	"&Unindent"
wxID_YES	"&Yes"
wxID_ZOOM_100	"&Actual Size"
wxID_ZOOM_FIT	"Zoom to &Fit"
wxID_ZOOM_IN	"Zoom &In"
wxID_ZOOM_OUT	"Zoom &Out"

wxBitmapButton

和普通按钮唯一不同的地方在于，它将显示一个小图片而不是标签文本。

下面演示了创建一个图片按钮的方法：

```
#include "wx/bmpbuttn.h"
wxBitmap bitmap(wxT("print.xpm"), wxBITMAP_TYPE_XPM);
wxBitmapButton* button = new wxBitmapButton(panel, wxID_OK,
    bitmap, wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW);
```

以及在Windows平台上的显示效果:



图片按钮通常只需要指定一个拥有透明背景的图片，wxWidgets将会在任何状态的时候都使用这个图片，不过如果你愿意，你可以给不同的按钮状态指定不同的图片，比如选中状态，释放按钮状态以及不可用状态等。

XPM图片格式是推荐的图片格式，因为它可以很容易的提供透明相关的信息以及可以被包含在C++代码中。不过加载别的格式的图片也是可以的，比如常见的JPEG,PNG,GIF以及BMP格式。

wxBitmapButton的窗口类型

wxBU_AUTODRAW	如果指定了这个类型，图片按钮将只会使用标签图片以系统默认的方式自动被绘制，它将包含边框和**3D**的外观。如果没有指定这个类型，图片按钮将按照不同状态提供的不同的图片来进行绘制**.** 这个类型仅适用于**Windows**和**Mac OS.**
wxBU_LEFT	图片左对齐. 在Mac OS上忽略这个类型.
wxBU_TOP	图片顶端对齐. Mac OS不适用.
wxBU_RIGHT	图片右对齐. Mac OS不适用.
wxBU_BOTTOM	图片底端对齐. Mac OS不适用.

wxBitmapButton事件

图片按钮的事件和wxButton完全相同.

wxBitmapButton的成员函数

SetBitmapLabel和GetBitmapLabel用来操作按钮的主要标签图片. 你还可以使用SetBitmapFocus, SetBitmapSelected, 和SetBitmapDisabled函数来设置按钮被激活，被按下以及被禁用状态下对应的图片。

SetDefault将按钮设置为其父窗口的默认按钮，这样用户按回车键就相当于用左键点击了这个按钮.

wxChoice

选择控件由一个只读的文本区域组成，这个区域的文本通过对一个附属的下拉列表框中值进行选择来进行赋值。这个下拉列表框是默认不可见的，只有在用户用鼠标点击下拉按钮以后才会显示。

创建一个选择控件的代码如下，其中的参数含义依次为：父窗口，标识符，位置，大小，窗口类型以及文本选项。

```
#include "wx/choice.h"
wxArrayString strings;
strings.Add(wxT("One"));
strings.Add(wxT("Two"));
strings.Add(wxT("Three"));
wxChoice* choice = new wxChoice(panel, ID_COMBOBOX,
    wxDefaultPosition, wxDefaultSize, strings);
```

在大多数平台上，除了文本区域的文本是只读的以外，选择控件和wxComboBox(如下图所示)是非常相似的。在GTK+平台上，选择控件是一个拥有一 个下拉菜单的按钮。你可以用过设置wxComboBox窗口的只读属性来模拟选择控件，以便能够使得在选项过多的时候使用滚动条。



wxChoice的窗口类型

wxChoice控件没有特别的窗口类型.

wxChoice的事件

wxChoice控件产生wxCommandEvent类型的事件，如下表所示：

VT_CHOICE(id,func)	用于处理wxEVT_COMMAND_CHOICE_SELECTED事件,当有用户通过列表选择某个选项的时候产生.

wxChoice的成员函数

wxChoice相关的成员函数都是前面介绍过的wxControlWithItems类的函数，如： Clear, Delete, FindString, GetClientData, GetClientObject, SetClientData, SetClientObject, GetCount, GetSelection, SetSelection, GetString, SetString, GetStringSelection, SetStringSelection, Insert, 和IsEmpty.

wxComboBox

ComboBox是一个单行编辑框和一个列表框的组合，用来允许用户以和下拉框中的文本没有关系的方式设置或者获取编辑框中的文本。其中 的单行文本域可以是只读的，在这种情况下就和选择控件非常相似了。和选择控件一样，通常列表框是隐藏的，除非用户单击了编辑框旁边的小按钮。这个控件的目 的在提供一种紧凑的方法给用户，使他们既可以自己输入文本，也可以很方便的选择预定义好的文本。

创建一个ComboBox的方法和创建选择控件的方法很类似，如下所示：

```
#include "wx/combobox.h"
wxArrayString strings;
strings.Add(wxT("Apple"));
strings.Add(wxT("Orange"));
strings.Add(wxT("Pear"));
strings.Add(wxT("Grapefruit"));
wxComboBox* combo = new wxComboBox(panel, ID_COMBOBOX,
    wxT("Apple"), wxDefaultPosition, wxDefaultSize,
    strings, wxCB_DROPDOWN);
```

wxComboBox的窗口类型

wxCB_SIMPLE	**列表框永远显示. 仅适用于Windows平台.**	****
wxCB_DROPDOWN	列表框下拉显示.	
wxCB_READONLY	和wxCB_DROPDOWN的含义相同，只不过编辑框的文本只能通过列表框进行选择，即使在应用程序的代码里，也不允许编辑框中的文本不存在于列表框内。	
wxCB_SORT	列表框中的选项自动按照子母顺序排序.	

wxComboBox的事件

wxComboBox产生的是wxCommandEvent类型的事件，如下表所示：

EVT_TEXT(id, func)	**用来处理wxEVT_COMMAND_TEXT_UPDATED事件,当编辑框内文本变化时产生.**
EVT_COMBOBOX(id, func)	用来处理wxEVT_COMMAND_COMBOBOX_SELECTED事件，当用户通过列表框选择一个选项的时候产生.

wxComboBox的成员函数

除了wxControlWithItems的成员函数以外，额外的成员函数还包括：

Copy函数将编辑框中的文本拷贝到剪贴板， Cut函数除了作Copy函数的动作，还将编辑框中的文本清空， Paste函数则把剪贴板内的文本复制到编辑框中。

GetInsertionPoint函数返回当前编辑框中插入点的位置(长整型),SetInsertionPoint则用来设置这个位置.SetInsertionPointEnd用来将其设置到编辑框末尾。

GetLastPosition返回编辑框中的最后位置。

GetValue函数用来返回编辑框中的文本,SetValue则用来设置它.如果combobox拥有wxCB_READONLY类型，则参数中的文本必须在列表框内，否则在发表版本中，这条语句将被忽略，而在调试版本中，则会出现一个告警。

SetSelection用来设置文本框中的一部分为选中状态，Replace则将文本框中的某一部分由给定的参数取代。Remove则移除文本框中的给定部分.

请同时参考wxControlWithItems类的这些成员函数: Clear, Delete, FindString, GetClientData, GetClientObject, SetClientData, SetClientObject, GetCount, GetSelection, SetSelection, GetString, SetString, GetStringSelection, SetStringSelection, Insert, 和IsEmpty.

wxCheckBox

CheckBox是一个通常拥有两种状态：选中或者未选中的控件。通常情况下，如果是选中的状态，则控件上显示一个小的叉号或者对号。通 常这个控件还包含一个标签，这个标签可以显示在控件的左边，也可以显示在控件的右边。CheckBox控件甚至还可以有第三个状态，姑且称之为混合状态或 者不确定状态。一个典型的用法是在安装程序中，对于某个组件来说除了要安装和不要安装两种状态以后，还可以有必须安装这种状态。

下面是创建CheckBox的例子代码：

```
#include "wx/checkbox.h"
wxCheckBox* checkbox = new wxCheckBox(panel, ID_CHECKBOX,
    wxT("&Check me"), wxDefaultPosition, wxDefaultSize);
checkBox->SetValue(true);
```

以及在windows平台上控件的样子：



以及另外的一个三态的wxCheckBox第三态的样子：



wxCheckBox的窗口类型

wxCHK_2STATE	创建一个二态选择框，这是默认类型.
wxCHK_3STATE	创建一个三态选择框.
wxCHK_ALLOW_3RD_STATE_FOR_USER	默认情况下，用户是不能设置第三种状态的，第三种状态只能在程序中通过代码来设置，使用这个类型可使得用户也可以通过鼠标设置第三态.
wxALIGN_RIGHT	使标签显示在选择框的左边.

wxCheckBox的事件

wxCheckBox产生wxCommandEvent类型的事件，如下表所示：

EVT_CHECKBOX(id, func)	用于处理 wxEVT_COMMAND_CHECKBOX_CLICKED事件, 当wxCheckBox的选择状态改变时产生.

wxCheckBox的成员函数

SetLabel和GetLabel用来设置选择框的标签文本. 在Windows和GTK+平台上，可以通过"&"前导字符设置快捷键.

GetValue和SetValue用来操作Bool型的当前选择状态. 使用Get3StateValue和Set3StateValue来操作三种状态wxCHK_UNCHECKED, wxCHK_CHECKED,或wxCHK_UNDETERMINED.

Is3State用于检测是否是三态选择框。

IsChecked用于检测当前是否为选中状态。

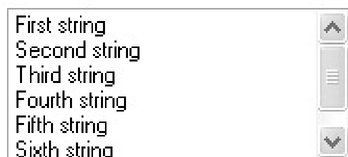
wxListBox 和 wxCheckListBox

wxListBox用来从一组基于0索引的字符串列表中选择一个或者多个。这一组备选的字符串列表显示在一个滚动窗口中，选中的文本被高 亮显示.列表框可以是单选的，这种情况下，如果一个选项被选中，以前被选中的选项就自动变为未选中状态，也可以是多选框，这种状态下，对某个选项的点击只 会导致这个选项的选中状态进行切换。

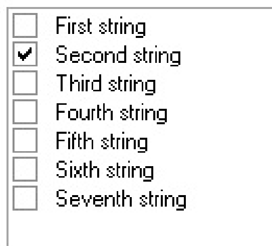
使用下面的代码创建一个列表框：

```
#include "wx/listbox.h"
wxArrayString strings;
strings.Add(wxT("First string"));
strings.Add(wxT("Second string"));
strings.Add(wxT("Third string"));
strings.Add(wxT("Fourth string"));
strings.Add(wxT("Fifth string"));
strings.Add(wxT("Sixth string"));
wxListBox* listBox = new wxListBox(panel, ID_LISTBOX,
    wxDefaultPosition, wxSize(180, 80), strings, wxLB_SINGLE);
```

在windows下的样子：

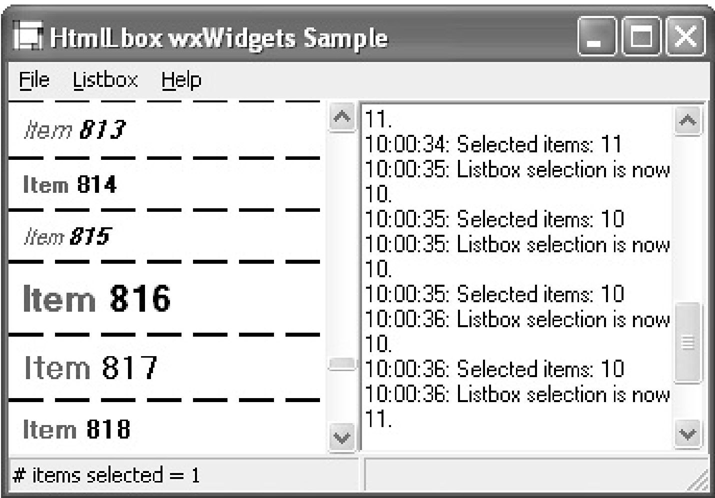


wxCheckListBox是wxListBox的派生类，继承了它的功能，另外它还在每个选项上额外显示一个复选框. 这个类包含在头文件wx/checklst.h中,下图演示了wxCheckListBox控件在windows上的样子：



如果有很多选项要显示，你可以考虑使用wxVListBox。这是一个虚的列表框类，它用来显示每个选项的方法是你在继承自这个类的派生类中实现的OnDrawItem函数和OnMeasureItem函数，而它的事件映射宏的格式则和wxListBox的一模一样。

wxHtmlListBox就是一个wxVListBox的派生类，它提供了显示复杂选项的一种简单的方法。你需要定义一个 wxHtmlListBox的派生类，然后在其OnGetItem函数中，为每个选项定义一段HTML文本，然后wxHtmlListBox则按照这段 HTML文本来显示各个选项。下图演示了光盘例子samples/htlbox在Windows Xp上的效果，在这个例子中，通过重载OnDrawSeparator函数实现不同选项的不同显示。



wxListBox和wxCheckListBox的窗口类型

wxLB_SINGLE	单选列表.
wxLB_MULTIPLE	多选列表.
wxLB_EXTENDED	扩展选择选项，用户可以通过Shift键或者鼠标以及其它一些键盘绑定进行快速多选.
wxLB_HSCROLL	创建水平滚动条如果选项的值过长. Windows only.
wxLB_ALWAYS_SB	总显示垂直滚动条.
wxLB_NEEDED_SB	只在需要的时候显示垂直滚动条.
wxLB_SORT	所有选项自动按照子母顺序排序.

wxListBox的wxCheckListBox事件

wxListBox和wxCheckListBox产生的事件类型wxCommandEvent类型的事件.

EVT_LISTBOX(id, func)	用于处理**wxEVT_COMMAND_LISTBOX_SELECTED**事件,当用户选择某个选项的时候产生**.**
EVT_LISTBOX_DCLICK(id, func)	用于处理wxEVT_COMMAND_LISTBOX_DOUBLECLICKED事件,当某个选项被双击的时候产生.
EVT_CHECKLISTBOX (id, func)	用于处理wxEVT_COMMAND_CHECKLISTBOX_TOGGLED事件,当wxCheckListBox的某个选项的选中状态发生改变的时候产生。

wxListBox 成员函数

Deselect不选则某个选项。 GetSelections使用wxArrayInt类型返回一组选中的索引。InsertItems用来插入一组选项，参数可以是个数和C++的wxString数组，以及插入点的组合，也可以是wxArrayString对象和插入点的组合.

Selected用来判断某个选项是否被选中。

Set用来清除选项并且用参数中的选项组重置选项。参数可以是个数和C++的wxString数组，以及插入点的组合，也可以是wxArrayString对象和插入点的组合.

SetFirstItem将某个选项设置为第一个可见的选项。

SetSelection和SetStringSelection用索引或者字符创值的方式指定某个选项的选中状态。

请同时参考wxControlWithItems的下列成员函数: Clear, Delete, FindString, GetClientData, GetClientObject, SetClientData, SetClientObject, GetCount, GetSelection, GetString, SetString, GetStringSelection, Insert, 和IsEmpty.

wxCheckListBox的成员函数

除了wxListBox的成员函数以外，wxCheckListBox还另外拥有下面的函数：

Check函数使用选项索引和一个bool值作为参数也控制选项的选中状态。

IsChecked用来判断某个选项是否为选中状态。

wxRadioBox

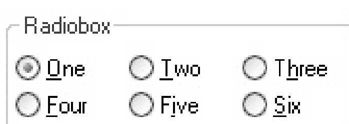
Radio Box用来在一组相关但是互斥的选项中进行选择。通常显示为一个可以拥有一个文本标签的静态框中的一组垂直的或者水平的选项按钮。

这些选项按钮的排列方式取决于构造函数中的两个参数，栏数和方向窗口有关的类型，方向有关的窗口类型包括wxRA_SPECIFY_COLS（默认值）和 wxRA_SPECIFY_ROWS，举例来说，如果你的radio box共有8个选项，栏数为2,方向为wxRA_SPECIFY_COLS，那么这些选项将会以两列四行的方式显示，而假如方向为 wxRA_SPECIFY_ROWS，则显示为四列两行。

下面演示了怎样创建一个有三栏的RadioBox:

```
#include "wx/radiobox.h"
wxArrayString strings;
strings.Add(wxT("&One"));
strings.Add(wxT("&Two"));
strings.Add(wxT("T&hree"));
strings.Add(wxT("&Four "));
strings.Add(wxT("F&ive "));
strings.Add(wxT("&Six "));
wxRadioBox* radioBox = new wxRadioBox(panel, ID_RADIOBOX,
    wxT("Radiobox"), wxDefaultPosition, wxDefaultSize,
    strings, 3, wxRA_SPECIFY_COLS);
```

以及它在windows系统中的样子：



wxRadioBox的窗口类型

wxRadioBox额外的窗口类型如下表所示：

wxRA_SPECIFY_ROWS	横向分栏.
wxRA_SPECIFIY_COLS	纵向分栏.

wxRadioBox事件

wxRadioBox产生wxCommandEvent类型的事件，如下表所示：

EVT_RADIOBOX(id, func)	用来处理wxEVT_COMMAND_RADIOBOX_SELECTED事件,当用户点击选项的时候产生.

wxRadioBox成员函数

Enable选中（或不选）某个选项.

FindString查找匹配的选项，返回这个选项的索引或者wxNOT_FOUND.

GetCount返回选项的总数。

GetString和SetString用来操作某个选项的标签文本. GetLabel和SetLabel则用来操作整个radio box的标签.

GetSelection返回基于0的选中的选项的索引. GetStringSelection则返回选中的选项的标签文本. SetSelection和SetStringSelection则用来设置对应的选项，通过这两个函数进行设置时不会发送任何事件。

Show函数用来显示或者隐藏某个特定的选项或者整个radio box控件。

wxRadioButton

一个radio按钮通常用来表示一组相关但是互斥的选项中的一个，它拥有一个按钮和一个文本标签，这个按钮的外观通常是一个圆形。

radio按钮有两种状态：选中和未选中。你可以创建一组radio按钮来代替前面介绍的radiobox控件，这样作的方法是，给第一个radio按钮 指定wxRB_GROUP类型，然后直到另外一个组被创建，或者没有同级的子控件的时候，这个组才结束，组中的控件既可以是radio按钮，也可以是别的 控件。

为什么要代替radiobox呢，其中一个原因是有时候你需要使用更复杂的布局，例如，你可能希望给每一个radio按钮增加一个额外的描述或者增加一个额外的窗口控件，又或者你不希望在一组radio周围显示一个静态的边框等。



下面是创建一组（两个按钮的代码）：

```
#include "wx/radiobut.h"
wxRadioButton* radioButton1 = new wxRadioButton (panel,
    ID_RADIOBUTTON1, wxT("&Male"), wxDefaultPosition,
    wxDefaultSize, wxRB_GROUP);
radioButton1->SetValue(true);
wxRadioButton* radioButton2 = new wxRadioButton (panel,
    ID_RADIOBUTTON2, wxT("&Female"));
//用于水平放置两个按钮的布局控件使用的代码
wxBoxSizer* sizer = new wxBoxSizer(wxHORIZONTAL);
sizer->Add(radioButton1, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
sizer->Add(radioButton2, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
parentSizer->Add(sizer, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
```

wxRadioButton的窗口类型

wxRB_GROUP	用来标识一组**radio**按钮的开始.
wxRB_USE_CHECKBOX	使用checkbox的按钮取代radio按钮(仅适用于Palm OS).

wxRadioButton的事件

wxRadioButton产生wxCommandEvent类型的事件，如下表所示：

EVT_RADIOBUTTON(id, func)	用于处理wxEVT_COMMAND_RADIOBUTTON_SELECTED事件，当用户点击某个radio按钮的时候产生。

wxRadioButton的成员函数

GetValue和SetValue使用bool类型来操作radio按钮的状态.

wxScrollBar

wxScrollBar用来单独的增加一个滚动条，这个滚动条和某些窗口自动增加的两个滚动条是有区别的，但是它们处理事件的方式是一样的。滚动条主要有下面四个属性：范围(range),滑块大小，页大小和当前位置。

范围的含义指的是和这个滚动条绑定的窗口的逻辑单位的大小。比如一个表格有15行，那么和这个表格绑定的滚动条的范围就可以设置为15。

滑块大小通常用来反映当前可视部分的大小，还用表格来作为例子，如果因为窗口大小的原因表格只能显示5行，那么滚动条的滑块大小就可以设置成5。如果滑块大小大于或者等于范围，在多数平台上，这个滚动条将自动隐藏。

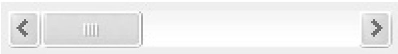
页大小指的是当滚动条执行翻页命令时需要滚动的单位数目。

当前位置指的是滑块当前所处的位置。

使用下面的代码来创建一个滚动条，其中构造函数的参数含义依次为父窗口，标识符，位置，大小和窗口类型：

```
#include "wx/scrolbar.h"
wxScrollBar* scrollBar = new wxScrollBar(panel, ID_SCROLLBAR,
    wxDefaultPosition, wxSize(200, 20), wxSB_HORIZONTAL);
```

在windows平台上，显示结果如下图所示：



创建一个滚动条以后，可以使用SetScrollbar函数来设置它的属性。前面在介绍wxScrolledWindow的时候已经介绍这个函数的用法。

wxScrollBar的窗口类型

wxSB_HORIZONTAL	指定滚动条为水平方向.
wxSB_VERTICAL	指定滚动条为垂直方向.

wxScrollBar的事件

wxScrollBar产生wxScrollEvent类型的事件。你可以使用EVT_COMMAND_SCROLL...事件映射宏加 窗口标识符来拦截由特定滚动条产生的相关事件，或者使用EVT_SCROLL...不带窗口标识符的事件映射宏来拦截除了本窗口以外来自其他的窗口的滚动 条事件。使用EVT_SCROLL(func)可以响应所有的滚动条事件。在附录I�事件类型和相关宏�中详细列举了所有的滚动事件，你也可以参考手册中 的相关内容。

wxScrollBar的成员函数

Getrange返回范围大小.

GetPageSize返回页大小.通常这个大小和滑块大小相同。

GetThumbPosition和SetThumbPosition用来操作滑块当前位置.

GetThumbLength返回滑块或者当前可是区域的大小.

SetScrollbar 用来设置滚动条的所有属性.比如滑块位置（逻辑单位）,滑块大小，范围，页大小以及一个可选的bool参数用来指示是否立即更新滚动条的显示。

wxSpinButton

wxSpinButton拥有两个小的按钮用来表示上下或者左右.这个控件通常和一个文本编辑框控件一起使用，以用来增加或者减少某个值。为了方便移植， 应该尽可能使用wxSpinCtrl来代替wxSpinButton,因为不是所有的平台都支持wxSpinButton.

这个控件（以及wxSpinCtrl）所支持的值的范围是平台相关的，但是至少在-32768到32768之间，

使用下面的代码来创建一个wxSpinButton,其中构造函数的参数的含义分别为：父窗口，标识符，位置，大小以及窗口类型：

```
#include "wx/spinbutt.h"
wxSpinButton* spinButton = new wxSpinButton(panel, ID_SPINBUTTON,
    wxDefaultPosition, wxDefaultSize, wxSP_VERTICAL);
```

在windows平台上，显示的结果如下图所示：



wxSpinButton的窗口类型

下表列出了wxSpinButton的窗口类型

wxSP_HORIZONTAL	**spin按钮为左右方向. 不支持wxGTK.**
wxSP_VERTICAL	spin按钮为上下垂直方向.
wxSP_ARROW_KEYS	用户可以使用方向键来改变相关值.
wxSP_WRAP	将最大值和最小值首尾相连，比如最小值的更小的下一个值将是最大值.

wxSpinButton的事件

wxSpinButton产生wxSpinEvent类型的事件, 如下表所示:

EVT_SPIN(id, func)	用来处理**wxEVT_SCROLL_THUMBTRACK**事件**,** 当上下（或左右）按钮被点击的时候产生**.**
EVT_SPIN_UP(id, func)	用来处理wxEVT_SCROLL_LINEUP事件,当向上（或者向左）的按钮被点击的时候产生.
EVT_SPIN_DOWN(id, func)	用来处理wxEVT_SCROLL_LINEDOWN事件,当向下（或者向右）按钮被点击的额时候产生.

wxSpinButton的成员函数

GetMax返回设置的最大值. GetMin返回设置的最小值. GetValue和SetValue用来操作当前值. SetRange用来设置最大和最小值.

wxSpinCtrl

wxSpinCtrl是wxTextCtrl和wxSpinButton控件的组合。当用户点击wxSpinButton的向上或者向下按钮的时候，wxTextCtrl中的值将会随之变化。用户也可以直接在wxTextCtrl中输入他想要的值。

下面的代码用来创建一个值范围在0到100之间，初始值为5的wxSpinCtrl。

```
#include "wx/spinctrl.h"
wxSpinCtrl* spinCtrl = new wxSpinCtrl(panel, ID_SPINCTRL,
    wxT("5"), wxDefaultPosition, wxDefaultSize, wxSP_ARROW_KEYS,
    0, 100, 5);
```

在windows平台上的显示效果如下图所示：



wxSpinCtrl的窗口类型

wxSP_ARROW_KEYS	用户可以通过方向键改变相关值.
wxSP_WRAP	将最大值和最小值首尾相连.

wxSpinCtrl事件

wxSpinCtrl产生wxSpinEvent类型的事件，如下表所示. 你也可以使用EVT_TEXT事件映射宏来处理其文本框的文本更新事件，处理函数的额参数类型为wxCommandEvent.

EVT_SPIN(id, func)	**Handles a wxEVT_SCROLL_THUMBTRACK event, generated whenever the up or down arrow is clicked.**
EVT_SPIN_UP(id, func)	Handles a wxEVT_SCROLL_LINEUP event, generated when the up arrow is clicked.
EVT_SPIN_DOWN(id, func)	Handles a wxEVT_SCROLL_LINEDOWN event, generated when the down arrow is clicked.
EVT_SPINCTRL(id, func)	Handles all events generated for the wxSpinCtrl.

wxSpinCtrl成员函数

GetMax返回设置的最大值. GetMin返回设置的最小值. GetValue和SetValue用来操作当前值. SetRange用来设置最大值和最小值.

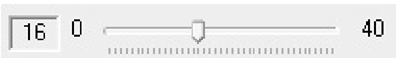
wxSlider

slider控件拥有一个滑条和一个滑块，可以通过移动滑条上的滑块来改变控件的当前值.

使用下面的代码创建一个取值范围为0到40,初始位置为16的wxSlider.

```
#include "wx/slider.h"
wxSlider* slider = new wxSlider(panel, ID_SLIDER, 16, 0, 40,
    wxDefaultPosition, wxSize(200, -1),
    wxSL_HORIZONTAL|wxSL_AUTOTICKS|wxSL_LABELS);
```

在windows平台上显示的外观如下所示：



wxSlider的窗口类型

wxSL_HORIZONTAL	显示水平方向的滑条.
wxSL_VERTICAL	显示垂直方向的滑条.
wxSL_AUTOTICKS	显示刻度标记.
wxSL_LABELS	显示最大、最小以及当前值的标签.
wxSL_LEFT	对于垂直滑条，将刻度显示在左面.
wxSL_RIGHT	对于垂直滑条，将刻度显示在右面.
wxSL_TOP	对于水平滑条，刻度显示在上面. 默认值为显示在底部.
wxSL_SELRANGE	允许用户通过滑条选择一个范围.仅适用于Windows.

wxSlider的事件

wxSlider产生wxCommandEvent类型的事件,如下表所示,但是如果你希望更好的控制，你可以使用EVT*COMMAND_SCROLL*...宏，其处理函数的参数类型为wxScrollEvent，参见附录I.

EVT_SLIDER(id, func)	用于处理wxEVT_COMMAND_SLIDER_UPDATED事件,当用户移动滑块的时候产生.

wxSlider的成员函数

ClearSel用来在设置了wxSL_SELRANGE类型的滑条上清除选择区域. ClearTicks则用来清除刻度，仅适用于windows系统.

GetLineSize和SetLineSize用来操作移动单位，这个移动单位用来在用户使用方向键操作滑块的时候使用. GetPageSize和SetPageSize也用来操作移动单位，这个单位在用户用鼠标点击滑条的任一边的时候使用.

GetMax返回当前设置的最大值.

GetMin返回当前设置的最小值.

GetSelEnd和GetSelStart用来返回选择区域的起点和终点; SetSelection用来设置选择区域的起点和终点. 这些函数都只适用于Windows.

GetThumbLength和SetThumbLength用来操作滑块的大小.

GetTickFreq和SetTickFreq用来操作滑条上刻度的密度.SetTick用来设置刻度的位置。这些函数仅适用于Windows.

GetValue返回滑条的当前值, SetValue用来设置滑条的当前值.

SetRange用来设置滑条的最大值和最小值.

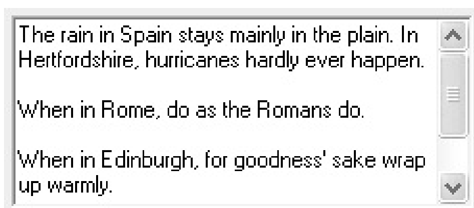
wxTextCtrl

文本控件是用来显示和编辑文本的控件，它支持单行和多行的文本编辑。在某些平台上，支持给文本控件中的文本设置一些简单的格式和风格。在windows平台，GTK+平台以及Mac OS X平台上通过使用wxTextAttr类来设置和获取文本的当前格式。

使用下面的代码创建一个支持多行文本的文本框控件：

```
#include "wx/textctrl.h"
wxTextCtrl* textCtrl = new wxTextCtrl(panel, ID_TEXTCTRL,
    wxEmptyString, wxDefaultPosition, wxSize(240, 100),
    wxTE_MULTILINE);
```

在windows平台上，多行文本框控件的外观如下:



多行文本框允许使用以"\n"分割的一组文本行的方式来处理一段文本，即使在非Unix的平台上，使用"\n"作为分割符也是允许的。这使得你可以忽略平 台之间换行符的差异。不过，作为代价，你将不能直接使用那些GetInsertionPoint函数或者GetSelection函数返回的索引，作为GetValue返回的字符串中的索引，因为在前者返回的索引中，操作系统可能会进行了一点点的偏移以便对应上平台使用的"\r\n"换行符，就象 windows平台上的那样。

如果你想从上面例子的函数的返回值中得到一个子字符串应该怎么办呢？你可以使用GetRange函数。它们返回的索引可以被用于它们自 己别的成员函数，比如SetInsertionPoint或者SetSelection。总而言之，不要将用多行文本框的成员函数返回的索引直接用于它的 内容字符串的索引，但是可以将其用于其它成员函数作为参数。

多行文本框支持设置文本格式：你可以为部分文本设置单独的文本颜色和字体。在windows平台上，这需要窗口使用wxTE_RICH 窗口类型。你可以在插入文本之前使用SetDefaultStyle函数，或者在插入文本以后使用SetStyle来改变已经存在于文本框中的文本的格 式。前者更有效率。

无论在哪种情况下，如果指定的格式中的部分格式是没有被指定的，则默认格式中相应的值将被使用，如果没有默认的格式，那个文本控件自己的属性将会被使用。

在下面的代码中，第二次调用SetDefaultStyle不会改变文本的前景颜色（仍然是红色），最后一次调用SetDefaultStyle则不会改变文本的背景颜色（仍然是灰色的）。

```
text->SetDefaultStyle(wxTextAttr(*wxRED));
text->AppendText(wxT("Red text\n"));
text->SetDefaultStyle(wxTextAttr(wxNullColour, *wxLIGHT_GREY));
text->AppendText(wxT("Red on gray text\n"));
text->SetDefaultStyle(wxTextAttr(*wxBLUE));
text->AppendText(wxT("Blue on gray text\n"));
```

wxTextCtrl的窗口类型

wxTE_PROCESS_ENTER	这个控件将会产生**wxEVT_COMMAND_TEXT_ENTER**事件**.**如果没有设置这个类型，回车键将会或者被空家内部处理，或者被**dialog**窗口用来遍历所有子窗口**.**
wxTE_PROCESS_TAB	这个控件将会在TAB键被按下的时候处理wxEVT_CHAR事件,否则TAB键用来在dialog的所有子窗口之间遍历.
wxTE_MULTILINE	支持多行文本.
wxTE_PASSWORD	文本将会以"*"显示.
wxTE_READONLY	文本只读.
wxTE_RICH	在Windows下使用富文本编辑控件. 这将允许控件存储超过64KB的文本;并且垂直滚动条自动在需要的时候显示.这个类型在别的平台上将被忽略.
wxTE_RICH2	在Windows下使用富文本编辑控件的2.0或者3.0版本; 垂直滚动条将始终被显示. 在其它平台忽略这个类型.
wxTE_AUTO_URL	高亮显示URL字符串，并且在其被点击的时候产生wxTextUrlEvents事件. 在windows平台上需要设置wxTE_RICH类型.仅适用于Windows和GTK+平台.
wxTE_NOHIDESEL	默认情况下，在windows平台上，如果文本框当前没有得到焦点，将不会高亮显示文本框中文本的选中部分，使用这个类型可以使其即使在文本框没有获得焦点的时候，被选部分也会高亮显示。其它平台则忽略这个类型。
wxHSCROLL	显示水平滚动条，这样就不需要文本自动换行了. 在GTK+平台上无效.
wxTE_LEFT	文本框中的文本左对齐 (默认).
wxTE_CENTRE	文本框中的文本居中对齐.
wxTE_RIGHT	文本框中的文本右对齐.
wxTE_DONTWRAP	等同于wxHSCROLL类型.
wxTE_LINEWRAP	如果文本行的长度超出可以显示的部分，则允许在文本行的任何位置换行，目前只支持wxUniversal版本.
wxTE_WORDWRAP	如果文本行的长度超出可以显示的部分，则允许在文本行的单词边界位置换行，目前只支持wxUniversal版本.
wxTE_NO_VSCROLL	Removes the vertical scrollbar. No effect on GTK+.

wxTextCtrl的事件

wxTextCtrl主要产生wxCommandEvent类型的事件，如下表所示：

EVT_TEXT(id, func)	用于处理**wxEVT_COMMAND_TEXT_UPDATED**事件,文本框内文本值改变的时候产生.
EVT_TEXT_ENTER(id, func)	用于处理wxEVT_COMMAND_TEXT_ENTER事件,在用户按下回车键的时候产生并且文本框设置了wxTE_PROCESS_ENTER窗口类型.
EVT_TEXT_MAXLEN(id, func)	用于处理wxEVT_COMMAND_TEXT_MAXLEN事件,当用户试图输入的文本长度超过SetMaxLength设置的长度的时候产生.仅适用于Windows和GTK+平台.

wxTextCtrl的成员函数

AppendText将文本添加在文本框最后, WriteText在当前的插入点插入文本. SetValue清除文本框中的当前文本然后赋值,赋值后IsModified函数返回False.在所有这些函数中，如果文本框支持多行文本，则可以使 用换行符。需要注意这些函数都会产生文本更新事件.

GetValue函数返回文本框的所有文本，如果是多行文本类型，则其中可以包含换行符. GetLineText返回其中一行. GetRange返回某个位置范围内的文本。

Copy函数拷贝选中的文本到剪贴板. Cut除了Copy以外还清除选中的文本. Paste则将剪贴板上的文本替换当前的选中文本，在用户界面刷新事件处理函数中，你还可以使用CanCopy, CanCut和CanPaste函数。

Clear清除所有文本.产生文本更新事件。

DiscardEdits复位内部的�已修改�标记，就好像文本框的文本已经被保存了那样。

EmulateKeyPress用来模拟按键输入，以便在文本框中进行某些修改.

GetdefaultStyle和SetDefaultStyle用来操作当前的默认文本格式. GetStyle返回某个位置文本的当前格式,SetStyle则用来设置某个范围内的文本格式e.

GetInsertionPoint和SetInsertionPoint函数用来操作新文本的插入点. GetLastPosition用来返回当前文本的最末位置,SetInsertionPointEnd用来将插入点设置在文本最末。

GetLineLength返回某个特定行的字符创长度。

GetNumberOfLines返回总行数。

GetStringSelection返回当前选中的文本。如果当前没有选中任何文本，则返回空字符串。GetSelection返回当前选中部分的索引. SetSelection则用两个整数参数来设置当前的选中部分。

IsEditable返回当前控件是否可以被编辑. SetEditable用来设置控件的可编辑状态以便让其只读或者可编辑. IsModified当文本框内的文本已经被编辑过的时候返回True. IsMultiline用来检测当前文本框是否是多行文本框。

LoadFile将文件内容读入文本框, SaveFile将文本框内容存入文件.

PositionToXY将象素值转换成文本的行号和位置, 而XYToPosition则刚好相反。

Remove删除给定区域的文本，. Replace则替换给定区域的文本。

ShowPosition使得文本控件显示包含给定位置的部分。

Undo撤消最近一次编辑, Redo重复最近一次编辑. 在某些平台上，这些操作可能什么也不作. 你可以用CanUndo和CanRedo来测试当前的平台是否支持撤消和重做动作。

wxToggleButton

wxToggleButton在用户点击以后保持按下状态.换句话说，除了长的象按钮，它其实更可以说是一个wxCheckBox.

创建wxToggleButton的代码如下:

```
#include "wx/tglbtn.h"
wxToggleButton* toggleButton = new wxToggleButton(panel, ID_TOGGLE,
    wxT("&Toggle label"), wxDefaultPosition, wxDefaultSize);
toggleButton->SetValue(true);
```

下图则显示了其在windows平台上的样子：



wxToggleButton的窗口类型

wxToggleButton没有特别的窗口类型.

wxToggleButton事件

wxToggleButton产生wxCommandEvent类型的事件。

EVT_TOGGLEBUTTON(id, func)	用于处理wxEVT_COMMAND_TOGGLEBUTTON_CLICKED事件, 用户点击该按钮的时候产生.

wxToggleButton的成员函数

SetLabel和GetLabel用来操作按钮上的标签. 在windows和GTK+平台上你可以使用"&"前导符来指定一个加速键.

GetValue和SetValue用来获取和设置按钮的状态。

4.7 静态控件

静态控件不响应任何用户输入，只用来显示一些信息或者增加应用程序的美感。

进度条

这是一个水平或者垂直的用来显示进度（通常是时间的进度）的控件。它不产生任何命令事件。下面的代码用来创建一个进度条：

```
#include "wx/gauge.h"
wxGauge* gauge = new wxGauge(panel, ID_GAUGE,
  200, wxDefaultPosition, wxDefaultSize, wxGA_HORIZONTAL);
gauge->SetValue(50);
```

在windows平台上的外观：



wxGauge的窗口类型

wxGA_HORIZONTAL	水平进度条.
wxGA_VERTICAL	垂直进度条.
wxGA_SMOOTH	创建一个光滑的进度条，进度条的每一段之间没有空格. 仅适用于Windows.

wxGauge事件

因为进度条只是用来显示信息，因此不产生任何事件。

wxGauge成员函数

GetRange和SetRange用来设置进度条的最大值。

GetValue和SetValue用来获取和设置进度条的当前值。

IsVertical用来检测是否是垂直进度条（否则就是水平的）。

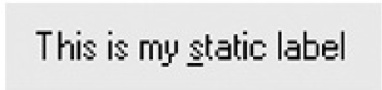
wxStaticText

静态文本控件用来显示一行或者多行的静态文本。

下面的例子创建了一个静态文本控件：

```
#include "wx/stattext.h"
wxStaticText* staticText = new wxStaticText(panel, wxID_STATIC,
  wxT("This is my &static label"),
  wxDefaultPosition, wxDefaultSize, wxALIGN_LEFT);
```

以及它在windows平台上的外观：



在静态文本控件标签中的前导符"&"，在某些平台（比如Windows和GTK+）上用来定义一个快捷键，通过这个快捷键可以直接访问到下一个非静态的控件。

wxStaticText的窗口类型

wxALIGN_LEFT	标签左对齐.
wxALIGN_RIGHT	标签右对齐.
wxALIGN_CENTRE	标签在水平方向上居中对齐.
wxST_NO_AUTORESIZE	默认情况下，静态文本控件会在调用SetLabel以后自动改变大小以使得其大小刚好满足标签文本的需要，如果设置了这个类型，则标签不会改变自己的大小。通常这个类型应该和上面的对齐类型一起使用因为如果没有设置这个类型，自动调整大小使得对齐没有任何意义。

wxStaticText的成员函数

GetLabel和SetLabel用户获取和设置文本标签。

wxStaticBitmap

静态图片控件显示一个图片。

使用下面的代码创建静态图片控件。

```
#include "wx/statbmp.h"
#include "print.xpm"
wxBitmap bitmap(print_xpm);
wxStaticBitmap* staticBitmap = new wxStaticBitmap(panel, wxID_STATIC,
  bitmap);
```

这会在作为父窗口的面板或者对话框上显示一个图片，如下图所示：



wxStaticBitmap的窗口类型

没有特别的窗口类型.

wxStaticBitmap的成员函数

GetBitmap和SetBitmap用来获取和设置其显示的图片。

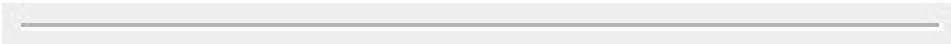
wxStaticLine

这个控件用来在其父窗口上显示一个水平或者垂直的长条，以便作为子窗口的静态分割条。

下面是创建wxStaticLine的代码：

```
#include "wx/statline.h"
wxStaticLine* staticLine = new wxStaticLine(panel, wxID_STATIC,
    wxDefaultPosition, wxSize(150, -1), wxLI_HORIZONTAL);
```

以及其在windows平台上的外观：



wxStaticLine的窗口类型

wxLI_HORIZONTAL	水平长条.
wxLI_VERTICAL	垂直长条.

wxStaticLine的成员函数

IsVertical用来检测是否为垂直长条.

wxStaticBox

这个控件用来在一组控件周围显示一个静态的拥有一个可选标签的矩形方框。到目前为止，这个控件不可以作为其它控件的父窗口。它围绕的那些控件是它的的兄弟窗口而非子窗口。它们应该在它后面创建，但是它们拥有同样的父窗口。在将来的版本中，也许会更改这个限制以便它可以同时容纳兄弟窗口和子窗口。

下面是创建一个wxStaticBox的例子代码：

```
#include "wx/statbox.h"
wxStaticBox* staticBox = new wxStaticBox(panel, wxID_STATIC,
  wxT("&Static box"), wxDefaultPosition, wxSize(100, 100));
```

以及它在windows平台上的样子：



wxStaticBox的窗口类型

没有特别的窗口类型

wxStaticBox的成员函数

GetLabel和SetLabel用来获取和设置其静态标签。

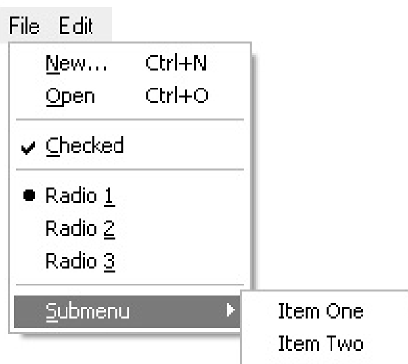
4.8 菜单

在这一小节中，我们来介绍一下怎样使用wxMenu,它用一种相对简单的办法来提供一组命令但是却不占用大量的空间。在下一小节，我们会描述一下怎样在菜单条上使用菜单。

wxMenu

菜单是指的一串命令，它可以从菜单条弹出，也可以从任何一个窗口上作为关联菜单，通过通常是右键单击来弹出。菜单项可以是一个普通的命令，也可以是一个复选框或者是一个单选框。菜单项可以被禁用，这是它将不能触发任何命令。某些菜单项可以通过一下特殊的三角符号来带出一个新的菜单，菜单中有可以有新的特殊菜单项，这种循环可以使用任意多次。另外一种特殊的菜单项是一个分割条，它只是简单的显示一行或者一段空白以便把两组菜单项进行分割。

下图显示了一个典型的拥有普通菜单项，复选框，单选框以及子菜单的菜单：



上面的例子还演示了两种快捷操作方法：加速键和快捷键。加速键是通过前导符"&"指定的，使用下划线表示，当菜单显示的时候可以通过这个键来执行相应的命令。而快捷键则是一个组合键，它可以在菜单没有显示的时候执行菜单命令，它通过一个TAB加一个组合键被定义。例如，上图中的 New菜单是通过下面的代码实现的：

```
menu->Append(wxID_NEW, wxT("&New...\tCtrl+N"));
```

更多通过菜单或者wxAcceleratorTable创建快捷键的方法将会在第6章说明。

菜单中的复选框和单选框的状态是由菜单类自动维护的。它们会在单击的时候自动将自己的状态改变，并且在下一次菜单展示的时候以新的状态展示。对于单选框来说，在改变自己状态的同时，它还会将其它同一组中的单选框更改为未选中的状态。你也可以在自己的代码中设置它们的状态（参考第9章）。

你可以通过wxWindow::PopupMenu函数将某个菜单在一个窗口的特定位置显示，比如下面的代码：

```
void wxWindow::OnRightClick(wxMouseEvent& event)
{
    if (!m_menu)
    {
        m_menu = new wxMenu;
        m_menu->Append(wxID_OPEN, wxT("&Open"));
        m_menu->AppendSeparator();
        m_menu->Append(wxID_EXIT, wxT("E&xit"));
    }
    PopupMenu(m_menu, event.GetPosition());
}
```

菜单事件在发送给它的父窗口以及进行其它的事件表匹配之前会首先发送给菜单自己。PopupMenu函数会导致程序短暂堵塞，直到用户关闭这个菜单。如果你愿意，你可以每次都释放旧的并且重新创建新的菜单，也可以每次都使用同一个菜单。

你应该尽可能的使用系统预定义的菜单标识符，比如wxID_OPEN, wxID_ABOUT, wxID_PRINT等等，在第三章中有一个完整的列表。需要特别指出的是，在Mac OS X上，标识符为wxID_ABOUT, wxID_PREFERENCES和 wxID_EXIT的菜单项不是显示在你定义的菜单中，而是显示在系统菜单中，wxWidgets自动为你的菜单作了这个调整。但是你需要注意在调整以后不要留下类似空的帮助菜单，或者是两个菜单分割条连在一起这样的不专业的情况出现。

在wxWidgets的发行版的例子中的samples/menu例子演示了所有菜单的功能，而在另外一个samples/ownerdrw例子中，则演示了怎样在菜单中使用定制的字体和图片。

wxMenu的事件

wxMenu相关的事件类型总共有四种: wxCommandEvent, wxUpdateUIEvent, wxMenuEvent, 和 wxContextMenuEvent.

下表列出了处理函数使用wxCommandEvent作为参数类型的事件映射宏。可用其来处理菜单命令，无论是弹出菜单命令还是主菜单（来自类似Frame窗口的菜单条上的菜单）命令。这种事件宏和工具条上的事件映射宏是一致的，这使得菜单和工具条上的按钮产生的事件可以通过同一个处理函数处理。

EVT_MENU(id, func)	用来处理**wxEVT_COMMAND_MENU_SELECTED**事件,某个菜单项被选中的时候产生.
EVT_MENU_RANGE(id1, id2, func)	用来处理wxEVT_COMMAND_MENU_RANGE事件,在某个范围内的菜单项被选中的时候产生.

下表列出了处理函数使用wxUpdateUIEvent作为参数类型的事件映射宏。这种宏对应的事件是在系统空闲的时候产生的，以便给应用程序一个更新用户界面的机会。例如，允许或者禁止一个菜单项等。尽管wxUpdateUIEvent可以被任何窗口产生，但是菜单产生的wxUpdateUIEvent还是有一些不同的地方，在于在这个事件中可以调用Check函数，SetText函数和Enable函数等。Check函数用来选中或者不选某个菜单项，而SetText函数用来设置菜单项的标签。如果菜单项的标签需要根据某种条件动态改变的话会比较有用。例如：

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_UPDATE_UI(ID_TOGGLE_TOOLBAR, MyFrame::OnUpdateToggleToolbar)
END_EVENT_TABLE()
void MyFrame::OnUpdateToggleToolbar(wxUpdateUIEvent& event)
{
    event.Enable(true);
    event.Check(m_showToolBar);
    event.SetText(m_showToolBar ?
                  wxT("Show &Toolbar (shown)") :
                  wxT("Show &Toolbar (hidden)"));
}
```

EVT_UPDATE_UI(id, func)	用来处理**wxEVT_UPDATE_UI**事件. 处理函数可以调用**Enable, Check,** 和**SetText**以及其它函数.
EVT_UPDATE_UI_RANGE(id1, id2, func)	用来处理wxEVT_UPDATE_UI事件，以便同时处理一组标识符.

关于界面更新的更多详情请参考第9章。

下表列出另外一些事件映射宏，其中EVT_CONTEXT_MENU的参数类型为wxContextMenuEvent，这是一个从 wxCommandEvent继承的事件类型，因此这个事件可以在父子窗口继承关系中传播。使用这个宏来拦截类似鼠标右键单击以产生关联菜单的事件，然后调用事件的GetPosition函数来获得菜单应该显示的准确位置。剩下的事件映射宏的处理函数采用wxMenuEvent对象作为参数，这些事件只从菜单条发送给其frame窗口，来告诉应用程序一个菜单已经被打开或者关闭了，或者某个菜单项正被高亮显示，默认的EVT_MENU_HIGHLIGHT 的处理函数是在主程序的状态栏显示这个菜单项的帮助信息，不过你可以提供你自己的处理函数来作一些不同的事情。

EVT_CONTEXT_MENU(func)	用来处理由于用户或者通过某个特殊按键（在**windows**平台上），或者通过单击鼠标右键产生的弹出一个上下文菜单的请求。处理函数的参数类型为**wxContextMenuEvent.**
EVT_COMMAND_CONTEXT_MENU(id, func)	和EVT_CONTEXT_MENU相似,不过多了一个窗口标识符参数.
EVT_MENU_OPEN(func)	用来处理wxEVT_MENU_OPEN事件,在一个菜单即将被打开的时候产生。在windows平台上主菜单上每次遍历只会导致一次这个事件产生.
EVT_MENU_CLOSE(func)	用来处理wxEVT_MENU_CLOSE事件,它在某个菜单被关闭的时候产生.
EVT_MENU_HIGHLIGHT(id, func)	用来处理wxEVT_MENU_HIGHLIGHT事件,当某个菜单项被高亮显示的时候产生，通常用来在主程序的状态栏上产生关于这个菜单项的帮助信息。
EVT_MENU_HIGHLIGHT_ALL(func)	用来处理wxEVT_MENU_HIGHLIGHT事件，在任何一个菜单项被高亮显示的时候产生。

wxMenu的成员函数

Append增加一个菜单项: 参数为标识符,菜单项标签, 帮助信息和菜单项类型 (wxITEM*NORMAL, wxITEM_SEPARATOR, wxITEM_CHECK* 或 *wxITEM* RADIO). 你也可以使用AppendCheckItem和AppendRadioItem来避免手动指定wxITEM_CHECK或wxITEM_RADIO参数类型.例如:

```
// 增加一个普通菜单项
menu->Append(wxID_NEW, wxT("&New...\tCtrl+N"));
// 增加一个复选框菜单项
menu->AppendCheckItem(ID_SHOW_STATUS, wxT("&Show Status"));
// 增加一个单选框菜单项
menu->AppendRadioItem(ID_PAGE_MODE, wxT("&Page Mode"));
Another overload of Append enables you to append a submenu, for example:
// 增加一个子菜单
menu->Append(ID_SUBMENU, wxT("&More options..."), subMenu);
```

还有一种Append的重载函数允许你直接使用一个wxMenuItem来增加一个菜单项，这是在菜单中增加图片或者使用自定义字体唯一的方法：

```
// 初始化图片和字体
wxBitmap bmpEnabled, bmpDisabled;
wxFont fontLarge;
// 创建一个菜单项
wxMenuItem* pItem = new wxMenuItem(menu, wxID_OPEN, wxT("&Open..."));
// 设置图片和字体
pItem->SetBitmaps(bmpEnabled, bmpDisabled);
pItem->SetFont(fontLarge);
// 增加到菜单中
menu->Append(pItem);
```

使用Insert函数来在特定的位置插入一个菜单项。使用Prepend, PrependCheckItem, PrependRadioItem和PrependSeparator在菜单最开始的地方插入一个菜单项。

AppendSeparator插入一个分割条, InsertSeparator在特定位置插入一个分割条.

```
// 插入一个分割条
menu->AppendSeparator();
```

Break函数在菜单里插入一个中断点，导致下一个插入的菜单项出现在另外一栏。

使用Check函数是标记复选框或者单选框的状态，参数为菜单项标识符和一个bool值。使用IsChecked函数来获取当前状态.

Delete函数删除并且释放一个菜单项使用菜单项标识符或者wxMenuItem指针。如果这个菜单项是一个子菜单，则子菜单将不被删除。如果你想删除并且释放一个子菜单，使用Destroy函数。Remove函数移除一个菜单项但是并不释放它，而是返回指向它的指针。

使用Enable函数来允许或者禁止一个菜单项。但是比直接调用这个方法更好的作法是处理用户界面更新事件（参考第9章）。IsEnabled函数返回当前的可用状态。

使用FindItem函数根据标签或者标识符找到一个菜单项，使用FindItemByPosition函数通过位置找到一个菜单项。

GetHelpString和SetHelpString函数用来访问菜单项的帮助信息.当菜单是菜单条的一部分时，高亮显示的菜单项的帮助信息将会显示在状态栏上。

GetLabel和SetLabel用来获取或者设置菜单项的标签.

GetMenuCount返回菜单项的个数。

GetMenuItems返回一个wxMenuItemList类型的菜单项的列表。

GetTitle和SetTitle用来获取或者设置菜单的可选标题，这个标题通常只在弹出菜单中有意义。

UpdateUI发送用户界面更新事件到其参数只是的事件处理对象，如果参数为NULL则发往菜单的父窗口.这个函数在菜单即将显示之前会被调用，但是应用程序也可以在自己认为需要的时候调用它。

4.9 控制条

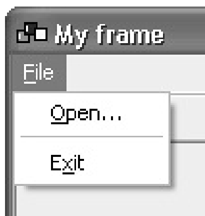
控制条提供了一个比较直观而方便的方法来排列多个控件。目前有三种类型的控制条，分别是：菜单条，工具条和状态条，其中菜单条只能在frame窗口上使用，工具条和状态条通常也是在frame窗口上使用，不过它们也可以作为别的窗口的子窗口。

wxMenuBar

菜单条包含一系列的菜单，显示在frame窗口顶部标题栏的下方。你可以通过调用SetMenuBar函数覆盖frame窗口当前的菜单条，使用下面的代码创建一个菜单条：

```
wxMenuBar* menuBar = new wxMenuBar;
wxMenu* fileMenu = new wxMenu;
fileMenu->Append(wxID_OPEN, wxT("&Open..."), wxT("Opens a file"));
fileMenu->AppendSeparator();
fileMenu->Append(wxID_EXIT, wxT("E&xit"), wxT("Quits the program"));
menuBar->Append(fileMenu);
frame->SetMenuBar(menuBar, wxT("&File"));
```

上述代码创建了一个只有一个菜单的菜单条，如下图所示：



你可以给菜单增加子菜单，也可以在菜单上增加单选框和复选框，还可以给菜单项增加快捷键和加速键（请参考本章前一章节的内容）。

如果你提供了一个帮助字符创，那么默认的EVT_MENU_HIGHLIGHT函数会将其显示到frame的窗口的状态栏上（如果有的话）。

wxMenuBar的窗口类型

wxMenuBar拥有一个wxMB_DOCKABLE类型，在GTK+平台上允许菜单条从主窗口分离出来。

wxMenuBar事件

参考本章前一小节中的相关内容。

wxMenuBar成员函数

Append函数在菜单条的末尾增加一个菜单项，一旦菜单项被成功增加，那么它将被菜单条管理并且在合适的时候被释放。这个函数参数为要增加的菜单以及一个标签。Insert函数则在给定的位置插入一个菜单。

Enable函数允许或者禁止给定标识符的菜单项. 使用IsEnabled函数判断菜单项是否被允许.

Check函数选中或者去选中一个复选框或者单选框菜单项. 使用IsChecked函数来获得当前的选择状态。

EnableTop函数允许或者禁止一整个菜单。参数为基于0的菜单索引。

FindMenu使用给定的标签字符串查找某个菜单，其中参数字符串可以带有前导字符也可以不带有前导字符.如果找不到则返回wxNOT_FOUND.

FindMenuItem通过菜单名和菜单项返回菜单项的索引。

FindItem通过给定的菜单项标识符返回wxMenuItem类型的菜单项，如果这个菜单项是一个子菜单，那么第二个参数将会返回这个子菜单的指针。

GetHelpString和SetHelpString用来获取或者设置某个菜单项的帮助信息。

GetLabel和SetLabel用来设置某个菜单项的标签,

GetLabelTop和 SetLabelTop用来获取和设置某个菜单在菜单条上的标签，参数为基于0的索引。

GetMenu根据索引返回对应菜单指针,

GetMenuCount返回菜单条上所有菜单的个数,

Refresh重画菜单条。

Remove移除一个菜单项并且返回对应的菜单指针，然后用户应该自己释放这个已经移除的菜单。

Replace函数则将某个位置的菜单使用新的菜单代替并且返回老的菜单的指针。 用户应该自己释放这个老的菜单,

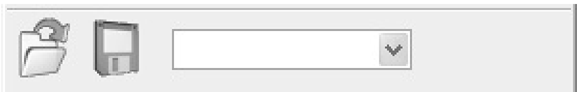
wxToolBar

工具条用来放置各种按钮和控件。工具条可以是水平的也可以是垂直的，其上的按钮可以是单选，复选以及push按钮。这些按钮可以显示标签也可以显示图片。如果你使用wxFrame::CreateToolBar函数创建工具条，或者使用wxFrame::SetToolBar函数将你创建的工具条和 frame窗口绑定，那么frame窗口将会管理这个工具条的位置和大小以及释放，并且工具条的大小也不算作frame的客户区大小。。任何其它方法创建的工具条，你都需要自己使用布局控件或者其它方法来来负责这个工具条的位置和大小,

下面是创建工具条以及和frame窗口绑定的例子代码：

```
#include "wx/toolbar.h"
#include "open.xpm"
#include "save.xpm"
wxToolBar* toolBar = new wxToolBar(frame, wxID_ANY,
    wxDefaultPosition, wxDefaultSize, wxTB_HORIZONTAL|wxNO_BORDER);
wxBitmap bmpOpen(open_xpm);
wxBitmap bmpSave(save_xpm);
toolBar->AddTool(wxID_OPEN, bmpOpen, wxT("Open"));
toolBar->AddTool(wxID_SAVE, bmpSave, wxT("Save"));
toolBar->AddSeparator();
wxComboBox* comboBox = new wxComboBox(toolBar, ID_COMBOBOX);
toolBar->AddControl(comboBox);
toolBar->Realize();
frame->SetToolBar(toolBar);
```

在windows平台上的外观如下图所示：



注意调用Realize之前，所有位于其上的按钮和控件应该已经被增加到了工具条中，否则工具条上将什么也不会显示。

你可以查看wxWidgets的samples/toolbar中的例子，来了解怎样更改工具条的方向，增加显示按钮上的标签以及更改按钮的大小等等其它方面的内容。

Windows平台上工具按钮上的图片的颜色

在windows平台上，wxWidgets试图将工具按钮上的图片的颜色映射到当前桌面风格下的标准颜色。通常来说，亮灰色（light gray）用来表示透明颜色。下表列出了所有这些颜色。事实上，工具按钮上的图片的颜色只需要接近于标准颜色。接近意味着RGB三原色的值和标准颜色的值差别在10个单位范围内。

颜色值	颜色名	用于
wxColour(0, 0, 0)	黑色	深色阴影
wxColour(128, 128, 128)	深灰	亮物体的3维边缘的阴面
wxColour(192, 192, 192)	亮灰	3维物体的表面(按钮背景), 表示透明区域
wxColour(255, 255, 255)	白色	亮物体的3维边缘的高亮面

对于16色的图片来说，这中映射是没有问题的，但是如果你使用的是颜色更丰富的按钮图片甚至真彩图片，那么这种映射可能会让你的按钮上的图片变的非常丑陋，在这种情况下，你可以使用下面的代码禁止这种映射：

```
wxSystemOptions::SetOption(wxT("msw.remap"), 0);
```

要使用上面的代码，你需要包含"wx/sysopt.h"头文件。

wxToolBar的窗口类型

wxTB_HORIZONTAL	创建水平工具条.
wxTB_VERTICAL	创建垂直工具条.
wxTB_FLAT	给工具条一个浮动外观. Windows and GTK+ only.
wxTB_DOCKABLE	使得工具条可以支持浮动和放置. GTK+ only.
wxTB_TEXT	显示按钮上的标签;默认情况下只显示图标.
wxTB_NOICONS	不显示按钮上的图标;默认情况下是显示的.
wxTB_NODIVIDER	指示工具条上没有分割线. Windows only.
wxTB_HORZ_LAYOUT	指示文本显示在图片的旁边而不是下面. 仅适用于Windows和GTK+. 这个类型必须和wxTB_TEXT共同使用.
wxTB_HORZ_TEXT	是wxTB_HORZ_LAYOUT和wxTB_TEXT的组合.

wxToolBar的事件

工具条事件映射宏如下表所示。象前面说过的那样，工具条的事件宏和菜单的事件宏是完全一样的，你既可以使用EVT_MENU宏，也可以使用EVT_TOOL宏。它们的事件处理函数的参数类型都是wxCommandEvent。对于它们中的大多数来说，窗口标识符指的都是工具按钮的窗口标识符，只有EVT_TOOL_ENTER事件宏的窗口标识符指的是工具条的窗口标识符，而对应的工具按钮的窗口标识符要从wxCommandEvent事件中获取，这是因为在表示鼠标离开一个工具按钮的时候，窗口标识符可能为-1,而在wxWidgets的事件体系中是不允许标识符为-1的。

EVT_TOOL(id, func)	用于处理**wxEVT_COMMAND_TOOL_CLICKED**事件 **(**和**wxEVT_COMMAND_MENU_SELECTED**的值相同**)**, 在用户点击工具按钮的时候产生**.** 在宏中使用工具按钮的标识符**.**
EVT_TOOL_RANGE(id1, id2, func)	用于处理一组工具按钮标识符的wxEVT_COMMAND_TOOL_CLICKED事件.
EVT_TOOL_RCLICKED(id, func)	用于处理wxEVT_COMMAND_TOOL_RCLICKED事件,用户在工具按钮上点击右键的时候产生.使用工具按钮的窗口标识符.
EVT_TOOL_RCLICKED_RANGE (id1, id2, func)	用于处理一组工具按钮标识符的wxEVT_COMMAND_TOOL_RCLICKED事件.
EVT_TOOL_ENTER(id, func)	用于处理wxEVT_COMMAND_TOOL_ENTER事件,用户的鼠标移入或者移出某个工具按钮的额时候产生，宏中使用工具条的窗口标识符，使用wxCommandEvent::GetSelection函数来获得移入的工具按钮的标识符，如果是移出则这个值是-1

wxToolBar的成员函数

AddTool增加一个工具按钮:指定按钮的标识符,一个可选的标签,一个图片,一个帮助信息,以及按钮的类型(wxITEM_NORMAL, wxITEM_CHECK, 或者wxITEM_RADIO). 使用InsertTool在某个特定的位置插入一个工具按钮. 也可以使用AddCheckTool和AddRadioTool来避免指定wxITEM_CHECK或wxITEM_RADIO类型. AddSeparator 增加一个分割线,基于实现的不同,它可能显示为一条线或者一段空白. 使用InsertSeparator在某个特定的位置插入一个分割线,例如下面的代码将增加一个复选框工具按钮，它包含一个标签("Save"), 一个图片，一个帮助字符串 ("Toggle button 1"):

```
toolBar->AddTool(wxID_SAVE, wxT("Save"), bitmap,
                 wxT("Toggle button 1"), wxITEM_CHECK);
```

AddControl增加一个控件，比如combo框. InsertControl在某个位置插入一个控件，

DeleteTool删除给定标识符的工具按钮. DeleteToolByPos删除指定位置的按钮. RemoveTool移除给定位置的那个按钮但是并不释放它，而是将它的指针返回给.

EnableTool用来允许或者禁止某个工具按钮,你可能想参考第9章中的方法，使用用户界面更新事件来更合理的作这个动作。 GetToolEnabled函数返回某个工具按钮的当前可用状态。

FindById和FindControl来通过窗口标识符寻找某个按钮或者某个控件。

如果你想使用非默认大小16x15的图标。你可以使用SetToolBitmapSize函数. 使用GetToolBitmapSize函数获得当前设置的图片尺寸. GetToolSize返回当前工具按钮整个的大小。

GetMargins和SetMargins用来获取和设置工具条的左右和上下的边界。

GetToolClientData和SetToolClientData通过窗口标识符获取和设置工具按钮绑定的某个继承自wxObject的类.

GetToolLongHelp和SetToolLongHelp用来获取或者设置和工具按钮绑定的长的帮助信息。这个信息可以被显示在状态条或者别的什么位置。GetToolShortHelp则用来获取或者设置工具按钮的短的帮助信息。

GetToolPacking 和 SetToolPacking用来获取和设置两排工具按钮之间的间隔。如果是垂直工具条，则为水平工具按钮之间的间隔，如果为水平工具条，则为垂直工具按钮之间的间隔。

GetToolPosition返回某个由窗口标识符指定的工具按钮在工具条上的位置。

GetToolSeparation和 SetToolSeparation用来获取或者设置工具条上分割线的大小。

GetToolState和SetToolState用来获取或者设置某个单选框或者复选框的状态。

Realize必须在任何按钮被增加以后调用。

ToggleTool反选某个单选或者复选按钮的状态。

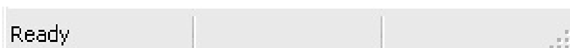
wxStatusBar

状态条是一个狭长的窗口，这个窗口通常被放置在一个Frame窗口的底部，用来提供一些状态信息。状态条可以包含一个或多个区域，区域可以拥有固定的或者可变的宽度。如果你使用wxFrame::CreateStatusBar或者wxFrame::SetStatusBar创建或者将某个状态条和frame窗口绑定，那么这个状态条的大小，位置以及其释放都由这个frame窗口负责，并且这个状态条的大小也不包含在frame窗口的客户区域内，反之，任何其它的方式创建的状态条，这些事情都要由你自己的代码去做。

下面是创建一个状态条的例子代码，这个状态条有三个区域，前两个的大小固定是60象素，第三个将使用剩余的区域：

```
#include "wx/statusbr.h"
wxStatusBar* statusBar = new wxStatusBar(frame, wxID_ANY,
    wxST_SIZEGRIP);
frame->SetStatusBar(statusBar);
int widths[] = { 60, 60, -1 };
statusBar->SetFieldWidths(WXSIZEOF(widths), widths);
statusBar->SetStatusText(wxT("Ready"), 0);
```

这段代码产生的结果如下图所示：



如果你想，你甚至可以在状态条的区域里面放置一些小的控件。这需要你自己管理它们的尺寸和大小，例如在你继承自wxStatusBar的新类的size事件的处理函数中。

wxStatusBar的窗口类型,注意你可以使用SetStatusStyles函数设置某个区域的类型

|:--- |:--- | | wxST_SIZEGRIP | 在状态条的右边显示一个小的修饰. |

wxStatusBar的事件

没有特别的事件。

wxStatusBar的成员函数

GetFieldRect返回某个区域的内部的大小和位置.

GetFieldsCount返回当前区域的个数. SetFieldsCount用来设置区域的个数。

GetStatusText返回当前某个区域的文本, SetStatusText用来设置某个区域的文本.

PushStatusText保存当前的区域的文本到一个堆栈中，并且把参数的字符串显示在状态条上. PopStatusText则将堆栈中最顶层的字符串显示在状态条上。

SetMinHeight设置状态条最小的合理高度。

SetStatusWidths采用一个整数数组来设置各个区域的宽度。其中整数代表绝对值，而负数则代表比例，比如说，如果你希望创建一个有三个区域的状态条，其中最右边的区域固定为100个象素，左边的两个区域按照2/3和1/3的比例瓜分剩下的区域，则你应该使用一个包含-2,-1,100三个整数的数组作为这个函数的参数。

SetStatusStyles使用区域个数和一组整数的类型数组来给各个区域设置不同的类型，这些整数的类型决定了区域的外观。其中 wxSB_NORMAL用来显示标准的拥有三维边界的下沉区域，而wxSB_FLAT显示一个没有边框的区域，而wxSB_RAISED则显示一个鼓起的拥有三维边框的区域。

第四章小结

这一章中，我们介绍了足够的关于各种窗口类和控件的知识，相信这足以是你可以开始构建你的各种应用程序。如果你还希望了解这个窗口或者控件的更多的知识，请参考wxWidgets的相关手册。对于那些更深入的窗口类，以及怎样创建自己的控件，我们将在第12章介绍。阅读wxWidgets发行版中的各个例子也将是非常有帮助的，比如samples/widgets, samples/toolbar, samples/text和 samples/listbox中的那些例子。

在接下来的一章里，我们将介绍一下应用程序怎样在不同的表面上作画，包括在窗口上，图片上或者是打印机上。

第五章绘画和打印

这一章里，我们来介绍一下设备上下文，也就是通常所说的在一个窗口或者一个打印页面上绘画的概念。我们将会讨论目前拥有的设备上下文类，以及 wxWidgets提供的和字体，颜色，线条以及填充等相关的绘画工具集。接下来我们会描述每个设备上下文的绘画函数以及wxWidgets支持打印的机制。在本章的最后，我们会简单讨论一下wxGLCanvas，这个类提供了一种在你的窗口中使用OpenGL技术绘制三维图形的方法。

5.1 理解设备上下文

在wxWidgets中，所有的绘画相关的动作，都是由设备上下文完成的。每一个设备上下文都是wxDC的一个派生类。从来就没有直接在窗口上绘画这种事情，每次在窗口上绘画，都要先创建一个窗口绘画设备上下文，然后在这个上下文上绘画。其它一些设备上下文是和bitmap图片或者打印机绑定的，你也可以设计自己的设备上下文。这样作一个最大的好处就是，你的绘画的代码是可以共享的，如果你的代码使用一个wxDC类型的参数，顶多再增加一个用于缩放的分辨率，那么这段代码就可以被同时用于在窗口绘制和在打印机上绘制。 让我们先来描述一下设备上下文的主要属性。

一个设备上下文拥有一个座标体系，它的原点通常在画布的左上角。不过这个位置可以通过调用SetDeviceOrigin函数改变，这样的效果就相当于对随后在这个上下文上的作的画进行平移。这种使用方法在对wxScrolledWindow进行绘画的时候非常常见。你还可以调用SetAxisOrientation来改变坐标系的放向，比如说你可以让Y轴是从下到上的放向而不是默认的从上到下的放向。

逻辑单位和设备单位是有区别的。设备单位是设备本地的单位，对于计算机屏幕来说，设备单位是一个象素，而对于一个打印机来说，设备单位是由它的分辨率决定的，这个分辨率可以用GetSize（用来取得设备单位的一页的大小）或者GetSizeMM(用来取得以毫米为单位的一页的大小)来获得。

设备上下文的映射模式定义了逻辑单位到设备单位的转换标准。要注意某些设备上下文（典型的例子比如：wxPostScriptDC）只支持wxMM_TEXT映射模式。下表列出了目前支持的映射模式：

wxMM_TWIPS	每个逻辑单位为**1/20**点, 或者**1/1440**英寸.
wxMM_POINTS	每个逻辑单位为1点, 或1/72英寸.
wxMM_METRIC	每个逻辑单位为1毫米.
wxMM_LOMETRIC	每个逻辑单位为1/10毫米.
wxMM_TEXT	每个逻辑单位为1象素.这是默认的映射模式.

你可以通过SetUserScale函数给逻辑单位指定一个缩放比例，这个比例乘以映射模式中指定的单位，可以得到实际逻辑单位和设备单位之间的关系，比如在wxMM_TEXT映射模式下，如果用户缩放比例为(1.0,1.0)，那么逻辑单位和设备单位就是一样的，这也是设备上下文中映射模式和用户缩放比例的缺省值。

设备上下文还可以通过SetClippingRegion函数指定一个区域，这个区域以外的部分将不被显示，可以使用 DestroyClippingRegion函数清除设备上下文当前指定的区域。举个例子，为了将一个字符串显示的范围限制在某个矩形区域以内，你可以先指定这个矩形区域为这个设备

上下文的当前区域，然后调用函数在这个设备上下文上绘制这个字符串，即使这个字符串很长，可能会超出这个矩形区域，超出的部分也将被截掉不被显示，然后再调用函数清除这个区域就可以了。

和现实世界一样，为了绘画，你需要先选择绘画的工具，如果你要画线，那么你需要选择画笔，要填充一个区域，你需要选择画刷，而当前字体，文本前景色和背景色等，则会决定你画布上的文本怎样显示。晚些时候我们会详细讨论这些。我们先来看看目前都支持那些设备上下文：

可用的设备上下文

下面列出了你可以使用的设备上下文:

- wxClientDC. 用来在一个窗口的客户区绘画。
- wxBufferedDC. 用来代替wxClientDC来进行双缓冲区绘画。
- wxWindowDC. 用来在窗口的客户区和非客户区（比如标题栏）绘画.这个设备上下文极少使用而且也不是每个平台都支持。
- wxPaintDC. 仅用在重绘事件的处理函数中，用来在窗口的客户区绘画。
- wxBufferedPaintDC. 和wxPaintDC类似，不过采用双缓冲区进行绘画。
- wxScreenDC. 用来直接在屏幕上绘画。
- wxMemoryDC. 用来直接在图片上绘画。
- wxMetafileDC. 用来创建一个图元文件(只支持Windows和Mac OS X).
- wxPrinterDC. 用来在打印机上绘画。
- wxPostScriptDC. 用来在PostScript文件上或者在支持PostScript的打印机上绘画。

接下来我们会描述一下怎样来创建和使用这些设备上下文。打印机设备上下文将会在本章稍后的“使用打印机的架构”小节中进行更详细的讨论。

使用wxClientDC在窗口客户区进行绘画

wxClientDC用来在非重绘事件处理函数中对窗口的客户区进行绘制。例如，如果你想作一个信手涂鸦的程序，你可以在你的鼠标事件处理函数中创建一个wxClientDC。这个设备上下文也可以用于擦除背景事件处理函数。

下面的例子演示了怎样使用鼠标在窗口上随便乱画：

```
BEGIN_EVENT_TABLE(MyWindow, wxWindow)
    EVT_MOTION(MyWindow::OnMotion)
END_EVENT_TABLE()
void MyWindow::OnMotion(wxMouseEvent& event)
{
    if (event.Dragging())
    {
        wxClientDC dc(this);
        wxPen pen(*wxRED, 1); // red pen of width 1
        dc.SetPen(pen);
        dc.DrawPoint(event.GetPosition());
        dc.SetPen(wxNullPen);
    }
}
```

在第19章，“使用文档和视图”中，实现了一个更具有可用性的涂鸦工具，它使用线段代替了点，并且实现了重做和撤消的操作。而且还存储了所有的线段，以便在windows需要重绘的时候重绘所有的线段。而使用上面的代码，你所作的画在下次windows重绘的时候将会消失。当然你还可以使用CaptureMouse和ReleaseMouse函数，以便在鼠标按下的时候，直接把鼠标限制在你的绘画窗口的客户区范围内。

一种替代wxClientDC的方法是使用wxBufferedDC，后者将你的所有绘画的结果保存在内存中，然后当自己被释放的时候一次性把所有的内容传输到窗口上。这样作的结果是使得整个绘画过程更平滑，如果你不希望用户看到绘画的结果一点一点的更新，你可以使用这种方法，使用 wxBufferedDC和使用wxClientDC的代码是完全一样的，另外为了提高效率，你可以在wxBufferedDC的构造函数中传递一个已经创建好的bitmap对象，以避免在每次创建wxBufferedDC的时候创建一个新的图片。

擦除窗口背景

窗口类通常会收到两种和绘画相关的事件：wxPaintEvent事件用来绘制窗口客户区的主要图形，而wxEraseEvent事件则用来通知应用程序擦除背景。如果你只拦截并处理了wxPaintEvent事件，则默认的wxEraseEvent事件处理函数会用最近一次的wxWindow::SetBackgroundColour函数调用设置的颜色或者别的合适的颜色清除整个背景。

这也许看上去有些混乱，但是这种把背景和前景分开的方法可以更好的支持那些使用这种架构的平台（比如windows）上的控件。举个例子，如果你希望在一个窗口上使用某种纹路的背景，如果你在OnPaint函数中是平铺纹理贴图，你可能会看到一些闪烁，因为在绘画之前，系统进行了清除背景的动作，在这种背景和前景分离的架构下，你可以拦截wxEraseEvent事件，然后在其处理函数中什么事情都不做，或者你干脆在擦除背景事件处理函数中平铺你的背景图，而在前景绘制事件的处理函数中绘制前景（当然，这样也会带来一些双缓冲绘画方面的问题，我们接下来会谈到）.

在某些平台上。仅仅是拦截wxEraseEvent事件仍然不足以阻止所有的系统默认的重绘动作，让你的窗口的背景不只是一个纯色背景的最安全的方法是调用wxWindow::SetBackgroundStyle然后传递wxBG_STYLE_CUSTOM参数。这将告诉wxWidgets把所有背景重绘的动作留给应用程序自己处理。

如果你真的决定实现自己的背景擦除事件处理函数，你可以先调用wxEraseEvent::GetDC来尝试返回一个已经创建的设备上下文，如果返回的值为空，再创建一个wxClientDC设备上下文。这将允许wxWidgets不在擦除背景事件中固定创建一个设备上下文，象前面例子中提到的那样，擦除背景事件处理函数中可能根本不会用到设备上下文，因此在事件中创建一个设备上下文可能是不必要的。下面的代码演示了怎样在擦除背景事件使用平铺图片的方法绘制窗口背景：

```
BEGIN_EVENT_TABLE(MyWindow, wxWindow)
  EVT_ERASE_BACKGROUND(MyWindow::OnErase)
END_EVENT_TABLE()
void MyWindow::OnErase(wxEraseEvent& event)
{
    wxClientDC* clientDC = NULL;
    if (!event.GetDC())
        clientDC = new wxClientDC(this);
    wxDC* dc = clientDC ? clientDC : event.GetDC() ;
    wxSize sz = GetClientSize();
    wxEffects effects;
    effects.TileBitmap(wxRect(0, 0, sz.x, sz.y), *dc, m_bitmap);
    if (clientDC)
        delete clientDC;
}
```

和重绘窗口事件一样，wxEraseEvent::GetDC返回的设备上下文已经设置了区域，使得只有需要重绘的部分才会被重绘。

使用wxPaintDC在窗口上绘画

如果你定义了一个窗口重画事件处理函数，则必须在这个处理函数中产生一个wxPaintDC设备上下文（即使你根本不使用它），并且使用它来进行你需要的绘画动作。产生这个对象将告诉wxWidgets的窗口体系这个窗口的需要重画的区域已经被重画了，这样窗口系统就不会重复的发送重画消息给这个窗口了。在重画事件中，你还可以调用wxWindow::GetUpdateRegion函数来获得需要重画的区域，或者使用wxWindow:: IsExposed函数来判断某个点或者某个矩形区域是否需要重画，然后优化代码使得仅在这个范围内的内容被重画，虽然在重画事件中创建的 wxPaintDC设备上下文会自动将自己限制在需要重画的区域内，不过你自己知道需要重画的区域的话，可以对代码进行相应的优化。

重画事件是由于用户和窗口系统的交互造成的，但是它也可以通过调用wxWindow::Refresh和wxWindow:: RefreshRect函数手动产生。如果你准确的知道窗口的哪个部分需要重画，你可以指定只重画那一部分区域以便尽可能的减少闪烁。这样作的一个问题是，并不能保证窗口在调用Refresh函数以后会马上重画。如果你真的需要立刻调用你的重画事件处理函数，比如说你在进行一个很耗时的计算，需要即时显示一些进度，你可以在调用Refresh或者RefreshRect以后调用wxWindow::Update函数。

下面的代码演示了如何在窗口正中位置画一个黑边红色的矩形区域，并且会判断这个区域是否位于需要更新的区域范围内以便决定是否需要重画。

```
BEGIN_EVENT_TABLE(MyWindow, wxWindow)
  EVT_PAINT(MyWindow::OnPaint)
END_EVENT_TABLE()
void MyWindow::OnPaint(wxPaintEvent& event)
{
    wxPaintDC dc(this);
    dc.SetPen(*wxBLACK_PEN);
    dc.SetBrush(*wxRED_BRUSH);
    // 获取窗口大小
    wxSize sz = GetClientSize();
    // 要绘制的矩形的大小
    wxCoord w = 100, h = 50;
    // 将我们的矩形设置在窗口正中,
    // 但是不为负数的位置
    int x = wxMax(0, (sz.xw)/2);
    int y = wxMax(0, (sz.yh)/2);
    wxRect rectToDraw(x, y, w, h);
    // 只有在需要的时候才重画以便提高效率
    if (IsExposed(rectToDraw))
        DrawRectangle(rectToDraw);
}
```

注意在默认情况下，当窗口大小改变时，只有那些需要重画的地方才会被更新，指定wxFULL_REPAINT_ON_RESIZE窗口类型可以覆盖这种默认情况以使得整个窗口都被刷新。在我们上面的例子中，就需要指定这种情况，因为我们矩形的位置是根据窗口大小计算出来的，如果窗口变大而我们只更新需要更新的部位，则可能在原来的窗口中留下半个矩形或者在屏幕上出现两个矩形，这和我们的初衷是不一致的。

wxBufferedPaintDC是wxPaintDC的双缓冲区版本。只需要简单的将重绘事件处理函数中的wxPaintDC换成 wxBufferedPaintDC就可以了，它会首先将所有的图片画在一个内存位图上，然后在自己被释放的时候一次性将其画在窗口上以减小闪烁。

正象我们前面提到的那样，另外一个减少闪烁的方法，是把背景和前景统一在窗口重画事件处理函数中，而不是将它们分开处理，配合 wxBufferedPaintDC，那么所有的绘画动作在完成之前都是在内存中进行的，这样在窗口被重绘之前你将看不到窗口背景被更新。你需要增加一个空的背景擦除事件处理函数，并且使用SetBackgroundStyle函数设置背景类型为wxBG_STYLE_CUSTOM以便告诉某些系统不要自动擦除背景。在wxScrolledWindow中你还有注意需要对绘画设备上下文的原点进行平移，并据此重新计算你自己的图片位置。（译者注：在 wxScrolledWindow窗口中的wxPaintDC的原点是当前窗口滚动位置下的原点，这通常不是我们所需要的，因为我们绘画通常要基于整个滚动窗口本身的原点，调用PrepareDC函数可以将其设置成滚动窗口本身的原点，在绘画的时候可以通过CalcUnscrolledPosition函数将当前客户区中的某个点转换成相对于整个滚动窗口区的座标，如下面的代码演示的那样）下面的代码演示了怎样在一个继承自 wxScrolledWindow的窗口类中进行绘画并且尽可能的避免出现闪烁。

```
#include "wx/dcbuffer.h"
BEGIN_EVENT_TABLE(MyCustomCtrl, wxScrolledWindow)
    EVT_PAINT(MyCustomCtrl::OnPaint)
    EVT_ERASE_BACKGROUND(MyCustomCtrl::OnEraseBackground)
END_EVENT_TABLE()
//重画事件处理函数
void MyCustomCtrl::OnPaint(wxPaintEvent& event)
{
    wxBufferedPaintDC dc(this);
    // 平移设备座标以便我们不需要关心当前滚动窗口的位置
    PrepareDC(dc);
    // 在重画绘制函数中绘制背景
    PaintBackground(dc);
    // 然后绘制前景
    ...
}
/// 绘制背景
void MyCustomCtrl::PaintBackground(wxDC& dc)
{
    wxColour backgroundColour = GetBackgroundColour();
    if (!backgroundColour.Ok())
        backgroundColour =
            wxSystemSettings::GetColour(wxSYS_COLOUR_3DFACE);
    dc.SetBrush(wxBrush(backgroundColour));
    dc.SetPen(wxPen(backgroundColour, 1));
    wxRect windowRect(wxPoint(0, 0), GetClientSize());
    //我们需要平移当前客户区矩形的座标以便将其转换成相对于整个滚动窗口而不是当前窗口的座标
    //因为在前面我们已经对设备上下文进行了PrepareDC的动作。
    CalcUnscrolledPosition(windowRect.x, windowRect.y,
                           & windowRect.x, & windowRect.y);
    dc.DrawRectangle(windowRect);
}
// 空函数 只为了防止闪烁
void MyCustomCtrl::OnEraseBackground(wxEraseEvent& event)
{
}
```

为了提高性能，当你使用wxBufferedPaintDC时，你可以维护一个足够大的（比如屏幕大小的）位图，然后将其传递给 wxBufferedPaintDC的构造函数作为第二个参数，这可是避免每次使用wxBufferedPaintDC的时候创建和释放一个位图。

wxBufferedPaintDC通常会从其缓冲区中拷贝整个客户区（用户可见部分）大小，在滚动窗口中，其内部创建的设备上下文并不会随着PrepareDC的调用平移其座标系。不过你可以通过在wxBufferedPaintDC的构造函数中指定 wxBUFFER_VIRTUAL_AREA（默认为wxBUFFER_CLIENT_AREA）参数来避免这一问题。不过这种情况下，你需要提供一个和整个滚动窗口的虚拟大小一样的缓冲区，而这通常效率是很低的，应该尽量避免。另外一个需要注意的是对于设置为wxBUFFER_CLIENT_AREA的 wxBufferedPaintDC到目前为止还不支持缩放（SetUserScale）。

你可以在随书光盘的examples/chap12/thumbnail例子的wxThumbnailCtrl控件中，找到使用wxBufferedPaintDC的完整的例子。

使用wxMemoryDC在位图上绘图

内存设备上下文是和一个位图绑定的设备上下文，在这个设备上下文上的所有绘画都将画在那个位图上面。使用的方法是先使用默认的构造函数创建一个内存设备上下文，然后使用SelectObject函数将其和一个位图绑定，在完成所有的绘画以后再调用SelectObject函数参数

为 wxNullBitmap来移除绑定的位图，代码如下所示：

```
wxBitmap CreateRedOutlineBitmap()
{
    wxMemoryDC memDC;
    wxBitmap bitmap(200, 200);
    memDC.SelectObject(bitmap);
    memDC.SetBackground(*wxWHITE_BRUSH);
    memDC.Clear();
    memDC.SetPen(*wxRED_PEN);
    memDC.SetBrush(*wxTRANSPARENT_BRUSH);
    memDC.DrawRectangle(wxRect(10, 10, 100, 100));
    memDC.SelectObject(wxNullBitmap);
    return bitmap;
}
```

你也可以使用Blit函数将内存设备上下文中的某一个区域拷贝到别的设备上下文上，在本章稍后的地方我们会对此进行介绍。

使用wxMetafileDC创建图元文件

wxMetafileDC适用于Windows和Mac OS X，它在这两个平台上分别提供了绘制Windows图元文件和Mac PICT的画布。允许在wxMetafile对象上绘画，这个对象保留了一组绘画动作记录，可以被其它应用程序使用或者通过wxMetafile:: Play函数将其绘制在别的设备上下文上。

使用wxScreenDC访问屏幕

使用wxScreenDC可以在整个屏幕的任何位置绘画。通常这在给拖放操作提供可见响应的时候比较有用（比如拖放分割窗口的分割条的时候）。处于性能方面的考虑。你可以将其操作的屏幕区域限制在一个矩形区域（通常是程序窗口所在的区域）内。当然，除了在屏幕上绘画，我们还可以把绘画从屏幕上拷贝到其它设备上下文中，以便实现屏幕捕获。因为无法限制别的应用程序的行为，所以wxScreenDC类通常在当前应用程序的窗口内工作的最好。

下面是将屏幕捕获到位图文件的例子：

```
wxBitmap GetScreenShot()
{
    wxSize screenSize = wxGetDisplaySize();
    wxBitmap bitmap(screenSize.x, screenSize.y);
    wxScreenDC dc;
    wxMemoryDC memDC;
    memDC.SelectObject(bitmap);
    memDC.Blit(0, 0, screenSize.x, screenSize.y, & dc, 0, 0);
    memDC.SelectObject(wxNullBitmap);
    return bitmap;
}
```

使用wxPrinterDC和wxPostScriptDC实现打印

wxPrinterDC用来实现打印机接口。在windows和Mac平台上，这个接口实现到标准打印接口的映射。在其它类Unix系统上，没有标准的打印接口，因此需要使用wxPostScriptDC代替（除非打开了Gnome打印支持，参考接下来的小节，“在类Unix系统上使用 GTK+进行打印”）。

可以通过多种途径创建wxPrinterDC对象，你还可以传递设置了纸张类型，横向或者纵向，打印份数等参数的 wxPrintData对象给它。一个简单的创建wxPrinterDC设备上下文的方法是显示一个wxPrintDialog对话框，在用户选择各种参数以后，使用wxPrintDialog::GetPrintDC的方法获取一个对应的wxPrinterDC对象。作为更高级的用法，你可以从 wxPrintout定义一个自己的派生类，以便更精确定义打印以及打印预览的行为，然后把它的一个实例传递给wxPrinter对象（在后面小节中还会详细介绍）。

如果你要打印的内容主要是文本，你可以考虑使用wxHtmlEasyPrinting类，以便忽略wxPrinterDC和 wxPrintout排版的细节：你只需要按照wxWidgets'实现的那些HTML的语法编写HTML文件，然后创建一个 wxHtmlEasyPrinting对象用来实现打印和预览，这通常可以节省你几天到几周的时候来对那些文本，表格和图片进行排版。详情请参考第12 章，“高级窗口类”。

wxPostScriptDC用来打印到PostScript文件。虽然这种方式主要应用在类Unix系统上，不过在别的系统上你一样可以使用它。用这种方式打印需要先产生PostScript文件，而且还要保证你拥有一个支持打印PostScript文件的打印机。

你可以通过传递wxPrintData参数，或者传递一个文件名，一个bool值以确定是否显示一个打印设置对话框和一个父窗口来创建一个wxPostScriptDC，如下所示：

```
#include "wx/dcps.h"
wxPostScriptDC dc(wxT("output.ps"), true, wxGetApp().GetTopWindow());
if (dc.Ok())
{
    // 告诉它在哪里找到AFM字体文件。
    dc.GetPrintData().SetFontMetricPath(wxGetApp().GetFontPath());
    // 设置分辨率（每英寸多少个点，默认720）
    dc.SetResolution(1440);
    // 开始绘画
    ...
}
```

wxPostScriptDC的一个特殊的地方在于你不能直接使用GetTextExtent来获取文本大小的信息。你必须先用 wxPrintData::SetFontMetricPath指定AFM（Adobe Font Metric）文件的路径，就象上面例子中的那样。你可以从下面的路径下载GhostScript AFM文件。

ftp://biolpc22.york.ac.uk/pub/support/gs_afm.tar.gz

5.2 绘画工具

在wxWidgets中，绘画操作就象是一个技术非常高超的艺术家，快速的选择颜色，绘画工具，然后画场景的一小部分，然后换不同的颜色，不同的工具，再绘画场景的其它部分，周而反复的操作。因此，接下来我们来介绍一下wxColour,wxPen, wxBrush, wxFont和wxPalette这些绘画工具。还有其它的一些内容也是有帮助的，比如wxRect，wxRegion，wxPoint和wxSize，我们会在第13章，�数据结构类�中对它们进行介绍。

注意这些类使用了�引用记数�的方法，使用内部指针以避免大块的内存拷贝，在大多数情况下，你可以直接以局部变量的方式定义颜色，画笔，画刷和字体对象而不用担心性能。如果你的程序确实因此而拥有性能上的问题，你才需要考虑采取一些方法来提高性能，比如将其中的一些局部变量改变成类的成员。

wxColour

你可以使用wxColour类来定义各种各样的颜色。（因为wxWidgets开始于爱丁堡，它的API使用英式拼写，不过对于那些对拼写很敏感的人，wxWidgets还是给wxColor定义了一个别名wxColour）。

你可以使用SetTextForeground和SetTextBackground函数来定义一个设备上下文中文本的颜色，也可以使用wxColour来创建画笔和刷子。

wxColour对象有很多种创建方法，你可以使用RGB三元色的值（0到255）来构建wxColour，或者通过一个标准的字符串，比如WHITE或者CYAN，或者从另外一个wxColour对象创建。或者你还可以直接使用系统预定的颜色对象指针： wxBLACK, wxWHITE, wxRED, wxBLUE, wxGREEN, wxCYAN,和 wxLIGHT_GREY.还有一个wxNullColour对象用来代表未初始化的颜色，它的Ok函数总是返回False。

使用wxSystemSettings类可以获取很多系统默认的颜色，比如3D表面颜色，默认的窗口背景颜色，菜单文本颜色等等。请参考相关文档中wxSystemSettings::GetColour的部分来获取详细的列表。

下面的例子演示了创建wxColour的方法：

```
wxColour color(0, 255, 0); // green
wxColour color(wxT("RED")); // red
// 使用面板的三维表面系统颜色
wxColour color(wxSystemSettings::GetColour(wxSYS_COLOUR_3DFACE));
```

wxTheColourDatabase指针用来在系统之类的颜色和颜色名之间建立映射，通过颜色名寻找对应的颜色对象或者通过颜色对象来寻找对应的颜色名，如下所示：

```
wxTheColourDatabase->Add(wxT("PINKISH"), wxColour(234, 184, 184));
wxString name = wxTheColourDatabase->FindName(
                                        wxColour(234, 184, 184));
wxString color = wxTheColourDatabase->Find(name);
```

下面列出了目前支持的标准颜色：aquamarine, black, blue, blue violet, brown, cadet blue, coral, cornflower blue, cyan, dark gray, dark green, dark olive green, dark orchid, dark slate blue, dark slate gray dark turquoise, dim gray, firebrick, forest green, gold, goldenrod, gray, green, green yellow, indian red, khaki, light blue, light gray, light steel blue, lime green, magenta, maroon, medium aquamarine, medium blue, medium forest green, medium goldenrod, medium orchid, medium sea green, medium slate blue, medium spring green, medium turquoise, medium violet red, midnight blue, navy, orange, orange red, orchid, pale green, pink, plum, purple, red, salmon, sea green, sienna, sky blue, slate blue, spring green, steel blue, tan, thistle, turquoise, violet, violet red, wheat, white, yellow, 和 yellow green.

wxPen

你可以使用SetPen函数指定一个设备上下文使用的画笔（wxPen）。画笔指定了随后的绘画操作中线条的颜色，粗细以及线条类型。wxPen的开销很小，你可以放心的在你的绘图代码中创建局部变量类型的画笔对象而不用对它们进行全局存储。

下表列出了目前支持的画笔线条类型，其中Hatch和stipple类型目前的GTK+版本不支持：

线形	示例	描述
wxSOLID	————————	纯色线.
wxTRANSPARENT	透明颜色.	
wxDOT	--------------	纯点线.
wxLONG_DASH	—— —— ——	长虚线.
wxSHORT_DASH	▬ ▬ ▬ ▬ ▬	短虚线.在windows平台上等同于wxLONG_SASH.
wxDOT_DASH	— - — - — -	短线和点间隔线.
wxSTIPPLE	★ ★ ★	使用一个位图代替点的点虚线,这个位图是其构造函数的第一个参数
wxUSER_DASH	—-—-—-—-—-—	自定义虚线. 参考用户手册.
wxBDIAGONAL_HATCH	////////////	反斜线虚线.
wxCROSSDIAG_HATCH	××××××××××	交叉虚线.
wxFDIAGONAL_HATCH	\\\\\\\\\\\\	斜线虚线.
wxCROSS_HATCH	++++++++++	十字虚线.
wxHORIZONTAL_HATCH	≡	水平线段虚线.
wxVERTICAL_HATCH	ıııııııııııı	垂直线段虚线.

使用SetCap定义粗线条的末端的样子：wxCAP_ROUND是默认的设置，只是粗线条的末端应该使用圆形，wxCAP_PROJECTING则只是使用方形并且有一个凸起，wxCAP_BUTT则只是直接使用方形。

使用SetJoin函数来设置当有线段相连时候的联结方式，默认的值是wxJOIN_ROUND，这种情况下转角是圆形的，其它可选的值还有wxJOIN_BEVEL和wxJOIN_MITER.

你也可以直接使用预定的画笔对象：wxRED_PEN, wxCYAN_PEN, wxGREEN_PEN, wxBLACK_PEN, wxWHITE_PEN, wxtrANSPARENT_PEN, wxBLACK_DASHED_PEN, wxGREY_PEN, wxMEDIUM_GREY_PEN 和 wxLIGHT_GREY_PEN.这些都是指针，所以在SetPen函数中使用的时候，应该使用"*"号指向它们的实例。还有一个预定义的对象（不是指针）wxNullPen，可以用来复位设备上下文中的画笔。

下面是创建画笔的一些演示代码,都用来产生一个纯红色的画笔：

```
wxPen pen(wxColour(255, 0, 0), 1, wxSOLID);
wxPen pen(wxT("RED"), 1, wxSOLID);
wxPen pen = (*wxRED_PEN);
wxPen pen(*wxRED_PEN);
```

上面例子中的最后两行使用了引用记数的方法，实际上内部指向同一个对象。这种引用记数的方法在绘画对象中很常用，它使得对象赋值和拷贝的系统开销非常小，不过同时它意味着一个对象的改变将会影响到其它所有使用同一个引用的对象。

一个既可以简化画笔对象的创建和释放过程，又不需要将画笔对象存储在自己的对象中的方法，是使用全局指针wxThePenList来创建和存储所有你需要的画笔对象，如下所示：

```
wxPen* pen = wxThePenList->FindOrCreatePen(*wxRED, 1, wxSOLID);
```

这个wxThePenList指向的对象将负责存储所有的画笔对象并且在应用程序退出的时候自动释放所有的画笔。很显然，你应该小心不要过量使用这个对象以免画笔对象占用大量的系统内存，而且也要注意前面我们提到过的使用引用对象的问题，你可以使用RemovePen函数从wxThePenList中删除一个画笔但是却不释放它所占的内存。

wxBrush

设备上下文当前使用的画刷对象可以用SetBrush函数指定，它决定设备上下文中图像的填充方式。你也可以使用它来定义设备上下文的默认背景，这样的定义方式可以使得背景不只是简单的纯色。和画笔对象一样，画刷对象的系统消耗也非常小，你可以直接使用局部变量的方式定义它。

画刷的构造函数采用一个颜色参数和一个画刷类型参数，如下表所示：

画刷类型	例子	描述
wxSOLID		纯色画刷.
wxTRANSPARENT	透明画刷.	
wxBDIAGONAL_HATCH		反斜线画刷.
wxCROSSDIAG_HATCH		交叉画刷.
wxFDIAGONAL_HATCH		斜线画刷.
wxCROSS_HATCH		十字画刷.
wxHORIZONTAL_HATCH		水平线画刷.
wxVERTICAL_HATCH		垂直线画刷.
wxSTIPPLE		位图画刷, 其位图在构造函数中指定.

你也可以直接使用下面这些系统预定义的画刷：wxBLUE_BRUSH, wxGREEN_BRUSH, wxWHITE BRUSH, wxBLACK_BRUSH, wxGREY_BRUSH, wxMEDIUM_GREY_BRUSH, wxLIGHT_GREY_BRUSH, wxtrANSPARENT_BRUSH, wxCYAN_BRUSH和wxRED_BRUSH. 这些都是指针，类似的还有wxNullBrush用来复位设备上下文的画刷。

下面是创建红色纯色画刷的例子：

```
wxBrush brush(wxColour(255, 0, 0), wxSOLID);
wxBrush brush(wxT("RED"), wxSOLID);
wxBrush brush = (*wxRED_BRUSH); // a cheap operation
wxBrush brush(*wxRED_BRUSH);
```

和画笔一样，画刷也有一个用来保存列表的全局指针指针： wxTheBrushList，你可以象下面这样使用它：

```
wxBrush* brush = wxTheBrushList->FindOrCreateBrush(*wxBLUE, wxSOLID);
```

同样要避免在应用程序过量使用以及要注意引用记数使用的问题。使用RemoveBrush来从wxTheBrushList中移除一个画刷而不释放其内存。

wxFont

你可以使用字体对象来设置一个设备上下文使用的字体，字体对象有下面一些属性：

字体大小用来以点（1/72英寸）为单位指定字体中的最大高度。wxWidgets会选择系统中最接近的字体。

字体家族用来指定一个家族系列，象下表中描述的那样，指定一个字体家族而不指定一个字体的名字是为了移植的方便，因为你不大可能要求某个字体的名字存在于所有的平台。

字体家族标识符	例子	描述
wxFONTFAMILY_SWISS	ABCDEFGabcdefg12345	非印刷字体，依平台的不同通常是Helvetica或Arial.
wxFONTFAMILY_ROMAN	ABCDEFGabcdefg12345	一种正规的印刷字体.
wxFONTFAMILY_SCRIPT	ABCDEFGabcdefg12345	一种艺术字体.
wxFONTFAMILY_MODERN	ABCDEFGabcdefg12345	一种等宽字体.通常是Courier
wxFONTFAMILY_DECORATIVE	ABCDEFGabcdefg12345	一种装饰字体.
wxFONTFAMILY_DEFAULT	wxWidgets选择一个默认的字体家族.	

字体类型可以是wxNORMAL, wxSLANT或wxITALIC。其中wxSLANT可能不是所有的平台或者所有的字体都支持。

weight属性的值则可以是wxNORMAL, wxLIGHT或wxBOLD.

下划线可以被设置或者关闭。

字体名属性是可选参数，用来指定一个特定的字体，如果其为空，则将使用指定字体家族默认的字体。

可选的编码方式参数用来指定字体编码和程序用设备上下文绘画的文本的编码方式的映射，详情请参考第16章，�编写国际化应用程序�。

你可以使用默认的构造函数创建一个字体，或者使用上表中列出的字体家族创建一个字体。

也可以使用下面这些系统预定义的字体： wxNORMAL_FONT, wxSMALL_FONT, wxITALIC_FONT和wxSWISS_FONT. 除了wxSMALL_FONT以外，其它的字体都使用同样大小的系统默认字体(wxSYS_DEFAULT_GUI_FONT), 而wxSMALL_FONT则比另外的三个字体小两个点.你可以使用wxSystemSettings::GetFont来获取当前系统的默认字体。

要使用字体，你需要在进行任何字体相关的操作（比如DrawText和GetTextExtent）之前，使用wxDC::SetFont函数设置字体。

下面的代码演示了怎样构造一个字体对象：

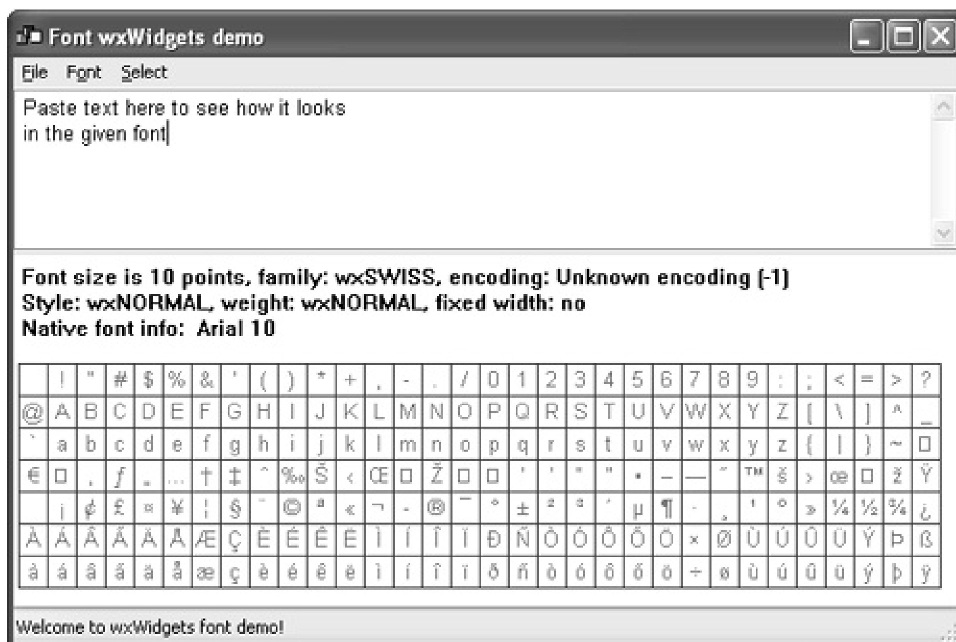
```
wxFont font(12, wxFONTFAMILY_ROMAN, wxITALIC, wxBOLD, false);
wxFont font(10, wxFONTFAMILY_SWISS, wxNORMAL, wxBOLD, true,
            wxT("Arial"), wxFONTENCODING_ISO8859_1));
wxFont font(wxSystemSettings::GetFont(wxSYS_DEFAULT_GUI_FONT));
```

和画笔和画刷一样，字体也有一个全局列表指针wxTheFontList，用来查找以前创建的字体或者创建一个新的字体：

```
wxFont* font = wxTheFontList->FindOrCreateFont(12, wxSWISS,
                                               wxNORMAL, wxNORMAL);
```

同样的，避免大量使用这个全局指针，因为其分配的内存要到程序退出的时候才会释放。你可以使用RemoveFont从中移除一个字体但是不释放相关的内存。

在本章晚些时候我们会看到一些使用文本和字体的例子。同时你也可以看一下字体例子程序，它允许你选择一个字体然后看看某些文本是什么样子，也可以让你更改字体的大小和别的属性。



wxPalette

调色板是一个表，这个表的大小通常是256,表中的每一个索引被映射到一个对应的rgb颜色值。通常在不同的应用程序需要在同一个显示设备上共享确定数目的颜色的时候使用。通过设置一个调色板，应用程序之间的颜色可以取得一个平衡。调色板也被用来把一个低颜色深度的图片映射到当前可用的颜色，因此每个wxBitmap都有一个可选的调色板。

因为现在大多数电脑的显示设备都支持真彩色，调色板已经很少使用了。应用程序定义的RGB颜色可以直接被显示设备映射到最接近的颜色。

创建一个调色板需要提供一个调色板大小参数和三个分别代表红绿蓝三种颜色的数组。你可以通过GetColoursCount函数得到当前调色板中条目的数量。GetRGB函数通过索引找到其代替的颜色的RGB值，而GetPixel则通过RGB值得到其相应的索引。

使用wxDC::SetPalette函数给某个设备上下文指定一个调色板。比如，你可以给当前的设备上下文指定一个位于某个低颜色深度的wxBitmap对象中的调色板，一边让设备上下文知道怎样把这个图片中的索引颜色映射到真实的RGB颜色。当在一个指定了调色板的设备上下文中使用 wxColour绘画的时候，系统会自动在设备上下文的调色板中查找最匹配的颜色的索引，因此你应该指定一个和你要用的颜色最接近的调色板。

wxPalette的另外一个用法是用来查询一个低颜色深度图像文件（比如GIF）中的图像的不同的颜色。如果这个图像拥有一个调色板，即时这个图片已经被转换成RGB格式，区分图像中的不同颜色也仍然是个很容易的事。类似的，通过调色板，你也可以把一个真彩色的图片转换成低颜色深度的图片，下面的代码演示了怎样将一个真彩色的PNG文件转换成8bit的windows位图文件：

```
// 加载这个PNG文件
wxImage image(wxT("image.png"), wxBITMAP_TYPE_PNG);
// 创建一个调色板
unsigned char* red = new unsigned char[256];
unsigned char* green = new unsigned char[256];
unsigned char* blue = new unsigned char[256];
for (size_t i = 0; i &lt; 256; i ++)
{
    red[i] = green[i] = blue[i] = i;
}
wxPalette palette(256, red, green, blue);
// 设置调色板和颜色深度
image.SetPalette(palette);
image.SetOption(wxIMAGE_OPTION_BMP_FORMAT, wxBMP_8BPP_PALETTE);
// 存储文件
image.SaveFile(wxT("image.bmp"), wxBITMAP_TYPE_BMP);
```

降低颜色深度的更实用的方法请参考第10章，�在程序中使用图片�中的�降低颜色深度�小节，介绍怎样用wxQuantize类来作这件事。

wxWidgets定义了一个空的调色板对象wxNullPalette.

(译者注：这一小节翻译的太费劲了)

5.3 设备上下文中的绘画函数

在这一小节里，我们来近距离的了解一下怎样在设备上下文中绘画。下表列出了设备上下文类主要的成员函数，在接下来的小节里，我们会举例介绍其中的大部分函数，你也可以从wxWidgets的手册中获取它们的详细信息。

Blit	把某个设备上下文的一部分拷贝到另外一个设备上下文上. 你可以指定的参数包括拷贝多大，拷贝到什么位置,拷贝的逻辑函数以及如果源设备上下文是内存设备上下文的时候，是不是使用透明遮罩等。
Clear	使用当前的背景刷刷新背景.
SetClippingRegion DestroyClippingRegion GetClippingBox	用来设置和释放设备上下文使用的区域。区域是用来将设备上下文中所有的绘画操作限制在某个范围内的，它可以是一个矩形，也可以是由wxRegion指定的复杂区域，使用GetClippingBox函数来取得包含当前区域的一个矩形范围。
DrawArc DrawEllipticArc	使用当前的画笔和画刷画一段圆弧或者椭圆弧线.
DrawBitmap DrawIcon	在指定位置画一副图片或者是一个图标.如果是图片可以指定一个透明遮罩.
DrawCircle	使用当前的画笔和画刷画一个圆形.
DrawEllipse	使用当前的画笔和画刷画一个椭圆形.
DrawLine DrawLines	使用当前的画笔画一条线或者多条线。最后的那个点是不被显示的。
DrawPoint	使用当前设置的画笔画一个点.
DrawPolygon DrawPolyPolygon	DrawPolygon函数通过指定一个点的数组或者指向点结构的指针的列表来绘制一个封闭的多边形，还可以指定一个可选的座标平移。wxWidgets 会自动封闭第一个点和最后一个点，所以你不必在最开始和最末端指定同一个点。DrawPolyPolygon函数则同时绘制多个多边形，在某些平台上这比多次调用DrawPolygon函数更有效率。
DrawRectangle DrawRoundedRectangle	使用当前的画笔和画刷绘制一个矩形或者圆角矩形
DrawText DrawRotatedText	使用当前字体，文本前景色和文本背景色在指定的位置绘制一段文本或者一段旋转的文本。
DrawSpline	使用当前的画笔在指定的控制点下使用云行规画一条平滑曲线.
FloodFill	以Flood填充的方式使用当前的画刷对指定起始点的范围进行填充（比如：封闭相邻同颜色区域中的颜色将被替换）.
Ok	如果设备已经准备好可以开始绘画则返回true.
	用来设置或者获取背景画刷设置。这些设置将在Clear函数或

SetBackground GetBackground	者其它一些设置了复杂的绘画逻辑参数的函数中被使用。默认设置为wxtrANSPARENT_BRUSH。
SetBackgroundMode GetBackgroundMode	用来设置绘制文本时候的背景类型，取值为wxSOLID或者wxTRANSPARENT。通常你希望设置为后者，以便保留绘制文本区域已经存在的背景。
SetBrush GetBrush	用来设置当前画刷，默认值未定义。
SetPen GetPen	用来设置当前画笔，默认值未定义。
SetFont GetFont	用来设置当前字体，默认值未定义。
SetPalette GetPalette	用来设置当前调色板。
SetTextForeground GetTextForeground SetTextBackground GetTextBackground	用来设置文本的前景颜色和背景颜色，默认值前景为黑色背景为白色。
SetLogicalFunction GetLogicalFunction	逻辑函数用来设置画笔或者画刷或者（在Blit函数中）设备上下文自己的象素的显示和传输方式。默认值wxCOPY表示直接显示或者拷贝当前颜色。
GetPixel	返回某个点的颜色.在wxPostScriptDC和wxMetafileDC中还没有实现这个功能.
GetTextExtent GetPartialTextExtents	返回给定文本的大小。
GetSize GetSizeMM	返回以设备单位或者毫米指定的长宽。
StartDoc EndDoc	这两个函数只在打印设备上下文中使用，前者显示一条消息表明正在打印，后者则隐藏这个消息。
StartPage EndPage	在打印设备上下文中开始和结束一页。
DeviceToLogicalX DeviceToLogicalXRel DeviceToLogicalY DeviceToLogicalYRel	将设备座标转换成逻辑座标，可以是绝对值也可以是相对值。
LogicalToDeviceX LogicalToDeviceXRel LogicalToDeviceY LogicalToDeviceYRel	将逻辑座标转换成设备座标。
SetMapMode GetMapMode	象前面描述过的那样，MapMode用来（和SetUserScale一起）指定逻辑单位到设备单位的映射。
SetAxisOrientation	用来指定X轴和Y轴的方向。默认X轴为从左到右（True），Y轴为从上到下（False）。
SetDeviceOrigin GetDeviceOrigin	设置座标原点，可以用来实现平移。
SetUserScale GetUserScale	设置缩放值。该值用于逻辑单位到设备单位的转换。

绘制文本

设备上下文绘制文本的方式取决于以下几个参数：当前字体，字体背景模式，文本背景色和文本前景色。如果背景模式是wxSOLID，文本背后的部分将会以当前的文本背景色擦除，如果是wxTRANSPARENT，则文本的背景将保留原先的背景。

传递给DrawText的参数是一个字符串和一个点（或者两个整数）。其中点（或者两个整数）指定的位置将会是文本最左上角的位置。下面是一个例子：

```
void DrawTextString(wxDC& dc, const wxString& text,
                    const wxPoint& pt)
{
    wxFont font(12, wxFONTFAMILY_SWISS, wxNORMAL, wxBOLD);
    dc.SetFont(font);
    dc.SetBackgroundMode(wxTRANSPARENT);
    dc.SetTextForeground(*wxBLACK);
    dc.SetTextBackground(*wxWHITE);
    dc.DrawText(text, pt);
}
```

你也可以使用DrawRotatedText函数来绘制一段旋转的文本，其中角度的值由函数的最后一个参数指定，下面的代码演示了一段以45度角增加的文本:

```
wxFont font(20, wxFONTFAMILY_SWISS, wxNORMAL, wxNORMAL);
dc.SetFont(font);
dc.SetTextForeground(wxBLACK);
dc.SetBackgroundMode(wxTRANSPARENT);
for (int angle = 0; angle &lt; 360; angle += 45)
    dc.DrawRotatedText(wxT("Rotated text..."), 300, 300, angle);
```

运行结果如下图所示：



在Windows平台上，只有TrueType类型的字体才可以实现旋转输出，要注意wxNORMAL_FONT指定的字体并不是TrueType字体。

通常情况下，你需要知道当前正要绘制的文本将占用设备上下文中多大的地方，你可以通过GetTextExtent函数来作到这一点，它的原型如下：

```
void GetTextExtent(const wxString& string,
    wxCoord* width, wxCoord* height,
    wxCoord* descent = NULL, wxCoord* externalLeading = NULL,
    wxFont* font = NULL);
```

从这个函数的原型中可以看出，后面的三个参数是可选的，其中descent参数和externalLeading参数（译者注：在汉字里面用处不大）的含义是：英文字符的基线通常不是字符的最底端，descent用来获取某个字符的最底端到基线的距离，而externalLeading则用来获取descent到下一行顶端的距离。最后一个参数font可以用来指定以这个字体为基准（而不是设备上下文自己的字体）来获取测量值。

下面的代码把一段文本在窗口的中间位置显示：

```
void CenterText(const wxString& text, wxDC& dc, wxWindow* win)
{
    dc.SetFont(*wxNORMAL_FONT);
    dc.SetBackgroundMode(wxTRANSPARENT);
    dc.SetTextForeground(*wxRED);
    // 获取窗口大小和文本大小
    wxSize sz = win->GetClientSize();
    wxCoord w, h;
    dc.GetTextExtent(text, & w, & h);
    // 计算为了居中显示需要的文本开始位置
    // 并保证其不为负数.
    int x = wxMax(0, (sz.x - w)/2);
    int y = wxMax(0, (sz.y - h)/2);
    dc.DrawText(msg, x, y);
}
```

如果你需要知道每个字符的精确占用大小，你可以使用GetPartialTextExtents函数，它使用wxString和一个wxArrayInt的引用作为参数。这个函数的效率在某些平台上优于对每个单个的字符使用GetTextExtent函数。

绘制线段和形状

这里用到的函数原型包括那些用来画点，画线，矩形，圆形以及椭圆等的函数。它们都使用当前的画笔设置和画刷设置，画笔用来决定轮廓线的颜色和模式，画刷用来决定填充的颜色和方式：

下面演示了一段代码和这段代码产生的图形：

```
void DrawSimpleShapes(wxDC& dc)
{
    // 设置黑色的轮廓线，绿色的填充色
    dc.SetPen(wxPen(*wxBLACK, 2, wxSOLID));
    dc.SetBrush(wxBrush(*wxGREEN, wxSOLID));

    // 画点
    dc.DrawPoint(5, 5);

    // 画线
    dc.DrawLine(10, 10, 100, 100);

    // 画矩形
    dc.SetBrush(wxBrush(*wxBLACK, wxCROSS_HATCH));
    dc.DrawRectangle(50, 50, 150, 100);

    // 改成红色画刷
    dc.SetBrush(*wxRED_BRUSH);

    // 画圆角矩形
    dc.DrawRoundedRectangle(150, 20, 100, 50, 10);

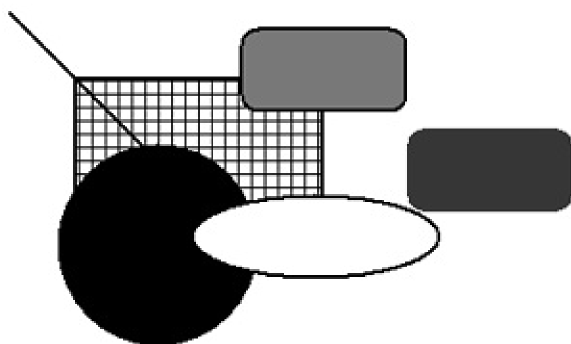
    // 没有轮廓线的圆角矩形
    dc.SetPen(*wxTRANSPARENT_PEN);
    dc.SetBrush(wxBrush(*wxBLUE));
    dc.DrawRoundedRectangle(250, 80, 100, 50, 10);

    // 改变颜色
    dc.SetPen(wxPen(*wxBLACK, 2, wxSOLID));
    dc.SetBrush(*wxBLACK);

    // 画圆
    dc.DrawCircle(100, 150, 60);

    // 再次改变画刷颜色
    dc.SetBrush(*wxWHITE);

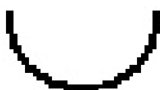
    // 画一个椭圆
    dc.DrawEllipse(wxRect(120, 120, 150, 50));
}
```



注意一个约定俗成的规矩，线上的最后一个点将不会绘制。

要绘制一段圆弧，需要使用DrawArc函数，提供一个起点，一个终点和一个圆心的位置。这段圆弧将以逆时针方向从起点画至终点，举例如下：

```
int x = 10, y = 200, radius = 20;
dc.DrawArc(xradius, y, x + radius, y, x, y);
```



绘制椭圆弧的函数DrawEllipticArc采用一个容纳这个椭圆的矩形的四个顶点，以及椭圆弧开始和结束的角度作为参数，如果开始和结束的角度相同，则将绘制一个完整的椭圆。

```
// 绘制一个包含在顶点为(10, 100),
// 大小为 200x40\. 圆弧角度从 270到420度的圆弧.
dc.DrawEllipticArc(10, 100, 200, 40, 270, 420);
```



如果你需要快速绘制很多条线段，那么DrawLines将会比多次调用DrawLine拥有更高的效率，下面的例子演示了快速绘制10个点之间的线段，并且指定了一个（100,100）的偏移量：

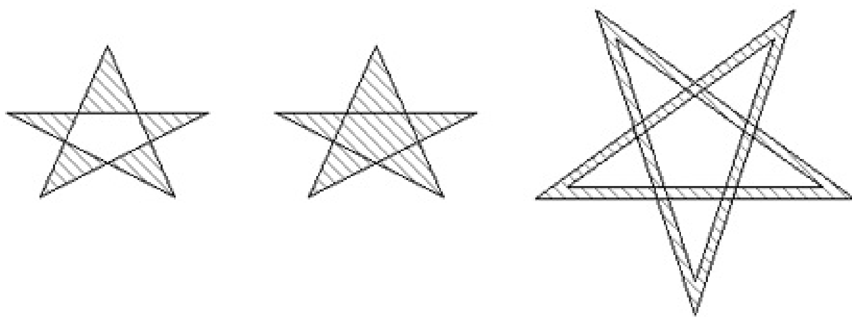
```
wxPoint points[10];
for (size_t i = 0; i &lt; 10; i++)
{
  pt.x = i*10; pt.y = i*20;
}
int offsetX = 100;
int offsetY = 100;
dc.DrawLines(10, points, offsetX, offsetY);
```

DrawLines不会填充线段环绕的区域。如果你想在画线的同时填充其环绕区域，你需要使用DrawPolygon函数，它的参数包括点的个数，点的列表，可选的平移参数以及填充类型。填充类型默认为wxODDEVEN_RULE，也可以使用wxWINDING_RULE。而DrawPolygonPolygon用来同时绘制多个Polygon，它的额外的一个参数是另外一个整数的数组，用来指定在前面的点的数组中每个 Polygon中点的个数。

下面代码演示了怎样使用这两个函数绘制polygons：

```
void DrawPolygons(wxDC& dc)
{
    wxBrush brushHatch(*wxRED, wxFDIAGONAL_HATCH);
    dc.SetBrush(brushHatch);
    wxPoint star[5];
    star[0] = wxPoint(100, 60);
    star[1] = wxPoint(60, 150);
    star[2] = wxPoint(160, 100);
    star[3] = wxPoint(40, 100);
    star[4] = wxPoint(140, 150);
    dc.DrawPolygon(WXSIZEOF(star), star, 0, 30);
    dc.DrawPolygon(WXSIZEOF(star), star, 160, 30, wxWINDING_RULE);
    wxPoint star2[10];
    star2[0] = wxPoint(0, 100);
    star2[1] = wxPoint(-59, -81);
    star2[2] = wxPoint(95, 31);
    star2[3] = wxPoint(-95, 31);
    star2[4] = wxPoint(59, -81);
    star2[5] = wxPoint(0, 80);
    star2[6] = wxPoint(-47, -64);
    star2[7] = wxPoint(76, 24);
    star2[8] = wxPoint(-76, 24);
    star2[9] = wxPoint(47, -64);
    int count[2] = {5, 5};
    dc.DrawPolyPolygon(WXSIZEOF(count), count, star2, 450, 150);
}
```

结果如下图所示：

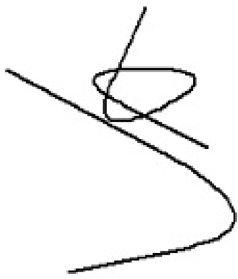


使用云行规画平滑曲线

DrawSpline函数让你可以绘制一个称为�云行规�的平滑曲线.这个函数有三个点和多个点两个形式，下面的代码对两者都进行了演示：

```
// 三点云行规曲线
dc.DrawSpline(10, 100, 200, 200, 50, 230);
// 五点云行规曲线
wxPoint star[5];
star[0] = wxPoint(100, 60);
star[1] = wxPoint(60, 150);
star[2] = wxPoint(160, 100);
star[3] = wxPoint(40, 100);
star[4] = wxPoint(140, 150);
dc.DrawSpline(WXSIZEOF(star), star);
```

结果如下图所示：



绘制位图

在设备上下文上绘制位图有两种主要的方法：DrawBitmap和Blit。DrawBitmap其实是Blit的一种简写形式，它使用一个位图，一个位置和一个bool类型的透明标志参数。根据图像的制作和读取过程的不同，位图的透明绘制可以通过指定一个透明遮罩或者一个Alpha通道（通常用来实现半透明）来实现，下面的代码演示了在一段文本上显示的半透明的图片：

```
wxString msg = wxT("Some text will appear mixed in the image's shadow...");
int y = 75;
for (size_t i = 0; i &lt; 10; i++)
{
    y += dc.GetCharHeight() + 5;
    dc.DrawText(msg, 200, y);
}
wxBitmap bmp(wxT("toucan.png"), wxBITMAP_TYPE_PNG);
dc.DrawBitmap(bmp, 250, 100, true);
```

运行结果如下图所示：文本在图片下面以半透明的方式隐隐浮现。

Blit函数就略微显的复杂些，它允许你拷贝一个设备上下文的一部分到另外一个设备上下文，下面是这个函数的原型：

```
bool Blit(wxCoord destX, wxCoord destY,
          wxCoord width, wxCoord height, wxDC* dcSource,
          wxCoord srcX, wxCoord srcY,
          int logicalFunc = wxCOPY,
          bool useMask = false,
          wxCoord srcMaskX = -1, wxCoord srcMaskY = -1);
```

这个函数将dcSource参数指定的设备上下文中开始于srcX，srcY的位置，大小为width，height的区域拷贝到目标设备上下文（函数调用者自己）的destX，destY的位置，并且使用指定的逻辑函数进行拷贝。默认的逻辑函数是wxCOPY，意味着直接把源中的象素原封不动的

传输到目标去。其它的逻辑函数依照平台的不同有所不同，不是所有的逻辑函数都支持所有的平台。我们将在本节稍后对逻辑函数进行专门介绍。

最后三个参数仅在园设备上下文为透明位图的时候才有效。useMask参数指定是否使用透明遮罩，而srcMaskX和srcMaskY则可以通过其设置不采用和主位图一致的遮罩位置。

下面的代码演示了怎样读取一个位图并将其平铺在另外一个更大的设备上下文上,并且保留图片本身的透明属性：

```
wxMemoryDC dcDest;
wxMemoryDC dcSource;
int destWidth = 200, destHeight = 200;
// 创建目标位图
wxBitmap bitmapDest(destWidth, destHeight);
// 加载调色板位图
wxBitmap bitmapSource(wxT("pattern.png"), wxBITMAP_TYPE_PNG);
int sourceWidth = bitmapSource.GetWidth();
int sourceHeight = bitmapSource.GetHeight();
// 用白色清除目标背景
dcDest.SelectObject(bitmapDest);
dcDest.SetBackground(*wxWHITE_BRUSH);
dcDest.Clear();
dcSource.SelectObject(bitmapSource);
// 将小的位图平铺到大的位图
for (int i = 0; i &lt; destWidth; i += sourceWidth)
    for (int j = 0; j &lt; destHeight; j += sourceHeight)
    {
        dcDest.Blit(i, j, sourceWidth, sourceHeight,
                    & dcSource, 0, 0, wxCOPY, true);
    }
//释放内存设备上下文的位图部分
dcDest.SelectBitmap(wxNullBitmap);
dcSource.SelectBitmap(wxNullBitmap);
```

你可以使用DrawIcon函数直接在设备上下文的某个位置显示图标，图标将总以透明方式显示：

```
#include "file.xpm"
wxIcon icon(file_xpm);
dc.DrawIcon(icon, 20, 30);
```

填充特定区域

FloodFill函数采用三个参数来填充某个特定的区域。一个起始点参数，一个颜色参数用来确定填充的边界和一个填充类型参数，设备上下文将使用当前的画刷定义进行填充。

下面的例子演示了绘制一个绿色矩形区域，它的轮廓线是红色，先进行黑色填充，进行蓝色填充：

```
// 画一个红边绿色的矩形
dc.SetPen(*wxRED_PEN);
dc.SetBrush(*wxGREEN_BRUSH);
dc.DrawRectangle(10, 10, 100, 100);
dc.SetBrush(*wxBLACK_BRUSH);
// 将绿色区域变成黑色
dc.FloodFill(50, 50, *wxGREEN, wxFLOOD_SURFACE);
dc.SetBrush(*wxBLUE_BRUSH);
// 开始填充蓝色(直到遇到红色)
dc.FloodFill(50, 50, *wxRED, wxFLOOD_BORDER);
```

如果找不到指定的颜色，这个函数可能返回失败，如果指定的点在当前区域以外，这个函数也将返回失败。这个函数不支持打印设备上下文以及wxMetafileDC。

逻辑函数

逻辑函数指定在绘画时，源象素怎样和目标象素进行合并操作，默认的wxCOPY只是使用源象素取代目标象素。其它的值则指定了一种逻辑操作。比如wxINVERT指定将目标象素的值取反作为新的值，这通常被用来绘制临时边框，因为使用这种方法进行绘图在第二次以同样的方法绘图的时候将会恢复原样。

下面的例子演示了怎样绘制一条点线段，然后再将相关区域恢复原来的样子。

```
wxPen pen(*wxBLACK, 1, wxDOT);
dc.SetPen(pen);
// 以取反逻辑函数绘制
dc.SetLogicalFunction(wxINVERT);
dc.DrawLine(10, 10, 100, 100);
// 再次绘制
dc.DrawLine(10, 10, 100, 100);
// 恢复正常绘制方法
dc.SetLogicalFunction(wxCOPY);
```

逻辑函数的另外一个用法是通过组合的方式创建新的图形。例如，我们可以用下面的方法来通过一个图片创建一组对应的拼图游戏需要的方块。首先在一幅白色背景的图片上使用固定但是随机的大小创建一个黑色轮廓线的网格作为拼图板的边界，然后对于每一个网格小块，使用flood-fill的方法将其填充成黑色以便创建一个白色背景上的黑色小方块，然后使用wxAND_REVERSE逻辑函数将原图Blit到这个模板，这样作的结果是使得方块内的部分变成原图，而白色的背景则变成黑色的背景，然后我们再指定黑色为透明色创建一个wxImage，然后将其转换成透明的wxBitmap对象，就可以直接在拼图游戏中使用了。（注意这样的作法需要原图中没有黑色，否则在拼图小方块中就会出现没有颜色的窟窿了）。

下表列出了所有的逻辑函数以及它们的含义：

逻辑函数	含义 **(src = 源, dst = 目的)**
wxAND	src AND dst
wxAND_INVERT	(NOT src) AND dst
wxAND_REVERSE	src AND (NOT dst)
wxCLEAR	0
wxCOPY	src
wxEQUIV	(NOT src) XOR dst
wxINVERT	NOT dst
wxNAND	(NOT src) OR (NOT dst)
wxNOR	(NOT src) AND (NOT dst)
wxNO_OP	dst
wxOR	src OR dst
wxOR_INVERT	(NOT src) OR dst
wxOR_REVERSE	src OR (NOT dst)
wxSET	1
wxSRC_INVERT	NOT src
wxXOR	src XOR dst

5.4 使用打印框架

我们已经介绍过，可以直接使用wxPrinterDC来进行打印。不过，一个更灵活的方法是使用wxWidgets提供的打印框架来驱动打印机。要使用这个框架，最主要的任务就是要实现一个wxPrintout的派生类，重载其成员函数以便告诉wxWidgets怎样打印一页（OnPrintPage），总共有多少页（GetPageInfo），进行页面设置(OnPreparePrinting)等等。而wxWidgets框架则负责显示打印对话框，创建打印设备上下文和调用适当的wxPrintout的函数。同一个wxPrintout类将被打印和预览功能一起使用。

当要开始打印的时候，一个wxPrintout对象实例被传递给wxPrinter对象，然后将调用Print函数开始打印过程，并且在准备打印用户指定的那些页面前显示一个打印对话框。如下面例子中的那样。

```
// 一个全局变量用来存储打印相关的设置信息
wxPrintDialogData g_printDialogData;
// 用户从主菜单中选择打印命令以后
void MyFrame::OnPrint(wxCommandEvent& event)
{
    wxPrinter printer(& g_printDialogData);
    MyPrintout printout(wxT("My printout"));
    if (!printer.Print(this, &printout, true))
    {
        if (wxPrinter::GetLastError() == wxPRINTER_ERROR)
            wxMessageBox(wxT("There was a problem printing.\nPerhaps your current printer
 is not set correctly?"), wxT("Printing"), wxOK);
        else
            wxMessageBox(wxT("You cancelled printing"),
                         wxT("Printing"), wxOK);
    }
    else
    {
        (*g_printDialogData) = printer.GetPrintDialogData();
    }
}
```

因为打印函数在所有的页面都已经被渲染和发送到打印机以后才会返回，因此wxPrintout对象可以以局部变量的方式创建。

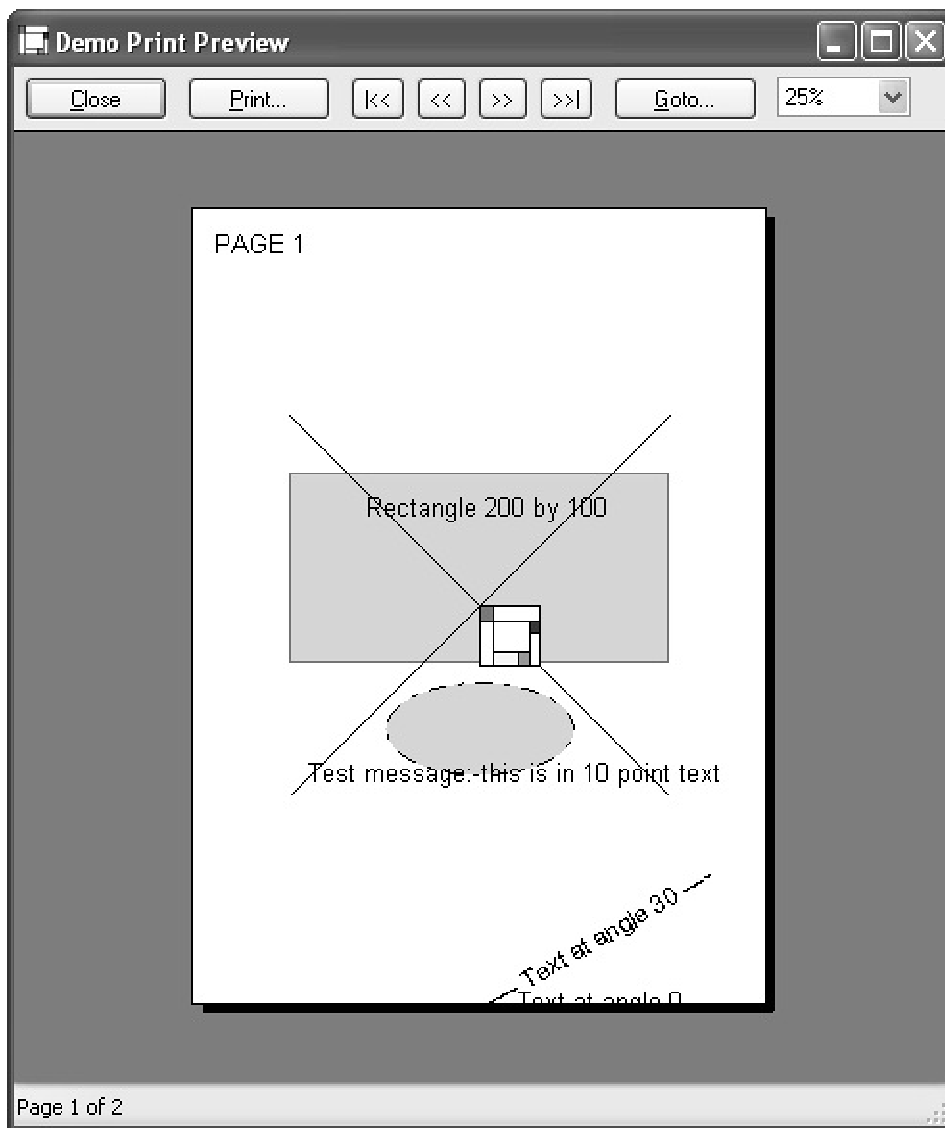
wxPrintDialogData对象用来存储所有打印有关的数据，比如用户选择的页面和份数等。将其保存在一个全局变量中并且在下次调用的时候传递给wxPrinter对象是一个不错的习惯，因此象上面代码中演示的那样，在创建wxPrinter的时候传递给它一份全局配置数据的指针，并在Print函数成功返回以后将当前的打印数据保存在全局变量里（当然，一个更专业的作法是将其保存在你的应用程序类中）。关于使用打印和页面设置对话框的详情请参考第8章，�使用标准对话框�。

如果要创建打印预览，你需要创建一个wxPrintPreview对象，给这个对象传递两个wxPrintout对象作为参数，一个用于预览，一个则用于在用户预览的时候直接请求打印，同样的，你可以传递一个全局的wxPrintDialogData对象给预览对象以便预览类使用用户以前选择的打印设

置数据。然后将这个预览类传递给wxPreviewFrame，然后调用wxPreviewFrame的Initialize函数和 Show函数显示这个窗口，如下所示：

```
// 用户选择打印预览菜单命令
void MyFrame::OnPreview(wxCommandEvent& event)
{
    wxPrintPreview *preview = new wxPrintPreview(
                                new MyPrintout, new MyPrintout,
                                & g_printDialogData);
    if (!preview->Ok())
    {
        delete preview;
        wxMessageBox(wxT("There was a problem previewing.\nPerhaps your current printer i
 not set correctly?"),
                     wxT("Previewing"), wxOK);
        return;
    }
    wxPreviewFrame *frame = new wxPreviewFrame(preview, this,
                                wxT("Demo Print Preview"));
    frame->Centre(wxBOTH);
    frame->Initialize();
    frame->Show(true);
}
```

当打印预览窗口被初始化的时候，它将禁用所有其它的顶层窗口以确保任何可能导致正在预览或者打印的文档内容发生改变的可能。关闭这个窗口将会导致两个wxPrintout对象自动被释放。下图显示了预览窗口的样子，可以看到它内含一个工具条，上面提供了页面遍历，打印以及缩放控制等功能。



关于wxPrintout的更多内容

当创建wxPrintout对象的时候，你可以传递一个可选的标题参数，在某些操作系统上，这个标题将显示在打印管理器上。另外，要重载一个 wxPrintout对象，你至少需要重载GetPageInfo, HasPage和OnPrintPage函数，当然，下面介绍的这些函数你都可以视需要进行重载。

首先来介绍GetPageInfo函数，它用来返回最小页码，最大页码，开始打印页码和结束打印页码。前两个参数用来提供一个可以打印的范围，后两个参数被设计来反应用户选择的页码范围，不过目前没有用处。最小页面的默认值为1,最大页码的默认值为32000,不过当HasPage函数返回False的时候，wxPrintout类也将停止打印。通常情况下，你的OnPreparePrinting应该计算当前打印内容在当前设置下的打印页数，将其保存在一个成员变量中，以便GetPageInfo函数返回正确的值，如下所示：

```
void MyPrintout::GetPageInfo(int *minPage, int *maxPage,
                             int *pageFrom, int *pageTo)
{
    *minPage = 1; *maxPage = m_numPages;
    *pageFrom = 1; *pageTo = m_numPages;
}
```

而HasPage函数则用来返回是否拥有某个页码，如果这个页码超出了最大页码的范围则必须返回False。通常你的实现类似下面的样子：

```
bool MyPrintout::HasPage(int pageNum)
{
    return (pageNum &gt;= 1 && pageNum &lt;= m_numPages);
}
```

OnPreparePrinting在预览或者打印过程刚开始的时候被调用，重载它使得应用程序可以进行各种设置工作，比如计算文档的总页数等。OnPreparePrinting可以调用wxPrintout的GetDC，GetPageSizeMM，IsPreview等函数，因此方便获取这些信息。

OnBeginDocument是在每次每篇文档即将开始打印的时候被调用的，如果这个函数被重载了，那么必须在重载的函数中调用 wxPrintout::OnBeginDocument。同样的wxPrintout::OnEndDocument也必须被它的重载函数调用。

OnBeginPrinting和OnEndPrinting函数则是在整个打印过程开始和结束的时候被调用，和当前打印多少份没有关系。

OnPrintPage会被传递一个页码参数，应用程序必须重载这个函数并且在成功打印这一页以后返回true。这个函数应该使用wxPrintout::GetDC来取得打印设备上下文已经进行绘画（打印）工作。

下面这些函数作为一些工具函数，你可以在你的重载函数中使用它们，而无需重载它们。

IsPreview函数用来检测当前正处于一个预览过程中还是一个真的打印过程中。

GetDC函数为当前正在进行的工作返回一个合适的设备上下文。如果是在真实的打印过程中，则返回一个wxPrinterDC，如果是预览，则返回一个wxMemoryDC，因为预览其实是在一个位图上通过内存设备上下文来渲染的。

GetPageSizeMM以毫米为单位返回当前打印页面的大小。GetPageSizePixels则以象素为单位返回这个值（打印机的最大分辨率）。如果是在预览过程中，这个大小和wxDC::GetSize返回的值通常是不一样大的（参见下面的说明），wxDC::GetSize返回用于预览的位图的大小。

GetPPIPrinter返回当前设备上下文每一个英寸对应的象素的数目，而GetPPIScreen则返回当前屏幕上每英寸对应的象素的数目。

打印和预览过程中的缩放

当你在窗口上绘画的时候，你可能不大关心图片的缩放，因为显示器的分辨率大部分都是相同的。然后，在面对一个打印机的时候，有几方面的因素可能导致你必须关心这个问题：

你需要通过缩放和重新放置图片的位置来保证图片位于某个页面之内，在某种情况下，你甚至可能需要把一个图片分成两半。

字体都是基于屏幕分辨率的，因此在打印文本的时候，你需要设置一个合适的缩放的值，以便使打印设备上下文符合屏幕的分辨率。在打印文本的时候，通过应该设置通过用GetPPIPrinter的值除以GetPPIScreen的值计算而得的值作为缩放因子的值。

当渲染预览图案的时候，wxWidgets使用了wxMemoryDC在一个位图上绘画。这个位图的大小（wxDC::GetSize）是基于当前预览的放大倍数的这就需要一个额外的缩放因子。通常这个缩放因子可以通过用GetSize返回的值除以GetPageSizePixels返回的实际页面的象素值来获得。通常这个值还应该乘以别的缩放因子定义的值。

你可以调用wxDC::SetUserScale来设置设备上下文的缩放因子，使用wxDC::SetDeviceOrigin来设置平移因子（例如，需要把一个图片放置在页面正中的时候）。如果有必要的话，你甚至可以在同一个页面绘画的时候反复使用不同的值来调用这两个函数。

wxWidgets的例子中的samples/printing演示了怎样在打印过程中使用缩放，下面列出的代码是其中进行缩放的适配代码，它将演示了在打印过程中和预览过程中将一幅大小为200x200象素的图片进行了缩放和放置，如下所示：

```
void MyPrintout::DrawPageOne(wxDC *dc)
{
    // 下面的代码可以这样写只是因为我们知道图片的大小是200x200
    // 如果我们不知道的话，需要先计算图片的大小
    float maxX = 200;
    float maxY = 200;
    // 让我们先设置至少50个设备单位的边框
    float marginX = 50;
    float marginY = 50;
    // 将边框的大小增加到图片的周围
    maxX += (2*marginX);
    maxY += (2*marginY);
    // 获取象素单位的当前设备上下文的大小
    int w, h;
    dc->GetSize(&w, &h);
    //计算一个合适的缩放值
    float scaleX=(float)(w/maxX);
    float scaleY=(float)(h/maxY);
    // 选择X或者Y方向上较小的那个
    float actualScale = wxMin(scaleX,scaleY);
    // 计算图片在设备上的合适位置以便居中
    float posX = (float)((w - (200*actualScale))/2.0);
    float posY = (float)((h - (200*actualScale))/2.0);
    // 设置设备平移和缩放
    dc->SetUserScale(actualScale, actualScale);
    dc->SetDeviceOrigin( (long)posX, (long)posY );
    // ok,现在开始画画
    dc.SetBackground(*wxWHITE_BRUSH);
    dc.Clear();
    dc.SetFont(wxGetApp().m_testFont);
    dc.SetBackgroundMode(wxTRANSPARENT);
    dc.SetBrush(*wxCYAN_BRUSH);
    dc.SetPen(*wxRED_PEN);
    dc.DrawRectangle(0, 30, 200, 100);
    dc.DrawText( wxT("Rectangle 200 by 100"), 40, 40);
    dc.SetPen( wxPen(*wxBLACK,0,wxDOT_DASH) );
    dc.DrawEllipse(50, 140, 100, 50);
    dc.SetPen(*wxRED_PEN);
    dc.DrawText( wxT("Test message: this is in 10 point text"),
                 10, 180);
}
```

在上面的例子中，我们只是简单的使用wxDC::GetSize来得到打印设备或者预览图像的分辨率，以便我们能够把要打印的图像放到合适的位置。我们没有关心类似每一个英寸多少个点这样的信息，因为我们不需要画精度很高的文本或者是线段。图片不需要很高的精度，因此我们只是进行简单的缩放以便它能够被合适的放置，不致于太大太小或者越界就可以了。

接下来，我们演示一下怎样进行精度很高的文本或者线段的打印以便看上去它和显示在屏幕上的是一致的。而不是只是进行简单的缩放：

```
void MyPrintout::DrawPageTwo(wxDC *dc)
{
    // 你可以使用下面的代码来设置打印机以便其可以反应出文本在屏幕上的大小
    // 另外下面的代码还将打印一个5cm长的线段。
    // 首先获得屏幕和打印机上各自的1英寸的逻辑象素个数
    int ppiScreenX, ppiScreenY;
    GetPPIScreen(&ppiScreenX, &ppiScreenY);
    int ppiPrinterX, ppiPrinterY;
    GetPPIPrinter(&ppiPrinterX, &ppiPrinterY);
    // 这个缩放因子用来大概的反应屏幕到实际打印设备的一个缩放
    float scale = (float)((float)ppiPrinterX/(float)ppiScreenX);
    // 现在，我们还需要考虑页面缩放
    // (比如：我们正在作打印预览，用户选择了一个缩放级别)
    int pageWidth, pageHeight;
    int w, h;
    dc->GetSize(&w, &h);
    GetPageSizePixels(&pageWidth, &pageHeight);
    // 如果打印设备的页面大小pageWidth == 当前DC的大小，就不需要考虑这方面的缩放了
    // 但是它们有可能是不一样的
    // 因此，缩放吧.
    float overallScale = scale * (float)(w/(float)pageWidth);
    dc->SetUserScale(overallScale, overallScale);
    // 现在我们来计算每个逻辑单位有多少个毫米
    // 我们知道1英寸大概是25.4毫米.而ppi
    // 代表的是以英寸为单位的. 因此1毫米就等于ppi/25.4个设备单位
    // 另外我们还需要再除以我们的缩放因子scale
    // (译者注：为什么这里是scale而不是overallScale？)
    // (其实overallScale比scale而言，多了一个预览的缩放)
    // （而在预览的时候，我们是希望线段的长度按照用户设置的比例变化的)
    // 因为我们的设备已经被设置了缩放因子
    // 现在让我们来画一个长度为50mm的L图案
    float logUnitsFactor = (float)(ppiPrinterX/(scale*25.4));
    float logUnits = (float)(50*logUnitsFactor);
    dc->SetPen(* wxBLACK_PEN);
    dc->DrawLine(50, 250, (long)(50.0 + logUnits), 250);
    dc->DrawLine(50, 250, 50, (long)(250.0 + logUnits));
    dc->SetBackgroundMode(wxTRANSPARENT);
    dc->SetBrush(*wxTRANSPARENT_BRUSH);
    dc->SetFont(wxGetApp().m_testFont);
    dc->DrawText(wxT("Some test text"), 200, 300 );
}
```

在类Unix系统上的GTK+版本上的打印

和Mac OS X以及Windows系统不同，Unix系统没有提供一个标准的API同时支持在屏幕上显示文本图片和在打印机上打印文本和图片。实际上，在类Unix系统中，屏幕显示是通过X11库（被GTK+封装又被wxWidgets封装）实现的，而打印要通过发送PostScript命令到打印机来完成。而这两种情况下使用不同的字体都是一个麻烦。直到最近，才有很少的程序在类Unix上提供了所见即所得的功能。以前wxWidgets提供自己的 PostScript实现，但是它很难和屏幕显示的内容完全一致。

从版本2.8开始，Gnome项目组开始通过libgnomeprint和libgnomeprintui库来提供打印支持，这样大多数的打印问题才算得以解决。从wxWidgets的版本2.5.4开始，GTK+的版本通过合适的配置以后可以支持这两个库。你需要使用--with- gnomeprint来配置wxWidgets，这将导致wxWidgets在运行期自动查找GNOME打印库。如果找的到，就使用它完成打印，否则就使用旧的PostScript打印的代码。需要说明的是，这并不需要用户的机器上一定要安装gnome的打印库程序才可以运行，因为程序本身并不依赖这些库。

5.5 使用wxGLCanvas绘制三维图形

感谢OpenGL和wxGLCanvas，让wxWidgets拥有了绘制三维图形的能力。如果你的平台不支持OpenGL，你仍然可以使用它的一个开放源码的实现Mesa。

要让wxWidgets在windows平台上支持wxGLCanvas，你需要编辑include/wx/msw/setup.h，设置 wxUSE_GLCANVAS为1，然后编译的时候在命令行使用USE_OPENGL=1，在连接的时候你也可能需要增加opengl32.lib。而在 Unix或者Mac OS X上，你只需要在配置wxWidgets的时候增加--with-opengl参数来打开OpenGL或者Mesa的支持。

如果你已经是一个OpenGL的程序员，那么使用wxGLCanvas是非常简单的。你只需要在一个frame窗口或者其他任何容器窗口内创建一个wxGLCanvas对象，然后调用wxGLCanvas::SetCurrent函数将OpenGL的命令指向这个窗口，执行OpenGL 命令，然后调用wxGLCanvas::SwapBuffers函数将当前的OpenGL缓冲区的内容绘制到窗口上。

下面的重绘事件处理函数演示了渲染一个三维立方体的一些基本代码书写原则。完整的例子可以在wxWidgets发行版本中的samples/opengl/cube目录中找到。

```
void TestGLCanvas::OnPaint(wxPaintEvent& event)
{
    wxPaintDC dc(this);
    SetCurrent();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-0.5f, 0.5f, -0.5f, 0.5f, 1.0f, 3.0f);
    glMatrixMode(GL_MODELVIEW);
    /* 清除颜色和深度缓冲 */
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    /* 绘制一个立方体的六个面 */
    glBegin(GL_QUADS);
    glNormal3f( 0.0f, 0.0f, 1.0f);
    glVertex3f( 0.5f, 0.5f, 0.5f); glVertex3f(-0.5f, 0.5f, 0.5f);
    glVertex3f(-0.5f,-0.5f, 0.5f); glVertex3f( 0.5f,-0.5f, 0.5f);
    glNormal3f( 0.0f, 0.0f,-1.0f);
    glVertex3f(-0.5f,-0.5f,-0.5f); glVertex3f(-0.5f, 0.5f,-0.5f);
    glVertex3f( 0.5f, 0.5f,-0.5f); glVertex3f( 0.5f,-0.5f,-0.5f);
    glNormal3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 0.5f, 0.5f, 0.5f); glVertex3f( 0.5f, 0.5f,-0.5f);
    glVertex3f(-0.5f, 0.5f,-0.5f); glVertex3f(-0.5f, 0.5f, 0.5f);
    glNormal3f( 0.0f,-1.0f, 0.0f);
    glVertex3f(-0.5f,-0.5f,-0.5f); glVertex3f( 0.5f,-0.5f,-0.5f);
    glVertex3f( 0.5f,-0.5f, 0.5f); glVertex3f(-0.5f,-0.5f, 0.5f);
    glNormal3f( 1.0f, 0.0f, 0.0f);
    glVertex3f( 0.5f, 0.5f, 0.5f); glVertex3f( 0.5f,-0.5f, 0.5f);
    glVertex3f( 0.5f,-0.5f,-0.5f); glVertex3f( 0.5f, 0.5f,-0.5f);
    glNormal3f(-1.0f, 0.0f, 0.0f);
    glVertex3f(-0.5f,-0.5f,-0.5f); glVertex3f(-0.5f,-0.5f, 0.5f);
    glVertex3f(-0.5f, 0.5f, 0.5f); glVertex3f(-0.5f, 0.5f,-0.5f);
    glEnd();
    glFlush();
    SwapBuffers();
}
```

下图演示了另外的一个OpenGL的例子，一个可爱的（当然，有点棱角的）企鹅，在例子程序中，你可以用鼠标来旋转它。完整的例子可以在光盘的samples/opengl/penguin目录中找到。



第五章小节

在本章中，你学习了怎样使用设备上下文进行绘画，怎样使用wxWidgets提供的打印框架，还看到了一个非常简单的使用wxGLCanvas的介绍。wxWidgets的发布包中有几个很本章内容相关的例子，列举如下以供参考。

- samples/drawing
- samples/font
- samples/erase
- samples/image
- samples/scroll
- samples/printing
- src/html/htmprint.cpp
- demos/bombs
- demos/fractal
- demos/life

对于更高级的二维绘画应用程序，你可能愿意考虑使用wxArt2D库，这个库提供了以SVG文件格式读取和保存图形对象，无闪烁更新，过渡色，矢量路径等等特性，附录E，“wxWidgets的第三方工具包”包含了获取wxArt2D的方法。

在下一章中，我们来看看wxWidgets怎样响应鼠标，键盘和游戏手柄输入。

第六章处理用户输入

所有的GUI程序都要以某种方式响应用户的输入，这一章我们来介绍以下在wxWidgets中怎样处理来自用户的鼠标，键盘以及游戏手柄的输入.

6.1 鼠标输入

总的说来，有两类的鼠标事件。基本的鼠标事件使用wxMouseEvent作为参数，不加任何翻译的发送给响应的窗口事件处理函数，而窗口事件处理函数则通常把它们翻译成对应的命令事件（wxCommandEvent）。

举例来说，当你在你的事件表中增加EVT_BUTTON事件映射宏的时候，它的处理函数的参数是一个由按钮产生的wxCommandEvent类型。而在控件内部，这个wxCommandEvent类型是按钮控件对EVT_LEFT_DOWN事件宏进行处理并将对应的鼠标事件翻译成wxCommandEvent事件的结果。（当然，在大多数平台上，按钮都是使用本地控件实现的，不需要自己处理底层的鼠标事件，但是对于别的定制的类来说，确是如此）

因为我们在前面已经介绍过处理命令事件的方法，我们将主要介绍怎样处理基本的鼠标事件。

你可以分别拦截鼠标左键，中键或者右键的鼠标按下，鼠标释放或者鼠标双击事件。你还可以拦截鼠标的移动事件（无论有没有鼠标按下）。你还可以拦截那些用来告诉你鼠标正在移入或者移出某个窗口的事件，最后，如果这个鼠标有滚轮，你还可以拦截鼠标的滚轮事件。

当你收到一个鼠标事件时，你可以获得鼠标按钮的状态信息，以及象Shift，Alt等等这些状态键的信息，你还可以获得鼠标指针相对于当前窗口客户区域左上角的座标值。

下表列出了所有对应的鼠标事件映射宏。需要注意的是，wxMouseEvent事件是不会传递给父窗口处理的，所以，为了处理这个事件，你必须重载一个新的窗口类，或者重载一个新的wxEvtHandler，然后将其挂载在某个窗口上，当然你还可以使用动态事件处理函数Connect，相关内容参见第三章。

EVT_LEFT_DOWN(func	用来处理**wxEVT_LEFT_DOWN**事件, 在鼠标左键按下的时候产生**.**
EVT_LEFT_UP(func)	用来处理wxEVT_LEFT_UP事件, 在鼠标左键被释放的时候产生.
EVT_LEFT_DCLICK(func)	用来处理wxEVT_LEFT_DCLICK事件,在鼠标左键被双击的时候产生.
EVT_MIDDLE_DOWN(func)	用来处理wxEVT_MIDDLE_DOWN事件, 在鼠标中键被按下的时候产生.
EVT_MIDDLE_UP(func)	用来处理wxEVT_MIDDLE_UP事件,当鼠标中键被释放的时候产生.
EVT_MIDDLE_DCLICK(func)	用来处理wxEVT_MIDDLE_DCLICK事件,在鼠标中键被双击的时候产生.
EVT_RIGHT_DOWN(func)	用来处理wxEVT_RIGHT_DOWN事件,鼠标右键被按下的时候产生.
EVT_RIGHT_UP(func)	用来处理wxEVT_RIGHT_UP事件,鼠标右键被释放的时候产生.
EVT_RIGHT_DCLICK(func)	用来处理wxEVT_RIGHT_DCLICK事件,鼠标右键被双击的时候产生.
EVT_MOTION(func)	用来处理wxEVT_MOTION事件,鼠标指针移动的时候产生.
EVT_ENTER_WINDOW(func)	用来处理wxEVT_ENTER_WINDOW事件,鼠标指针移入某个窗口的时候产生.
EVT_LEAVE_WINDOW(func)	用来处理wxEVT_LEAVE_WINDOW事件,鼠标移出某个窗口的时候产生.
EVT_MOUSEWHEEL(func)	用来处理wxEVT_MOUSEWHEEL事件,鼠标滚轮滚动的时候产生.
EVT_MOUSE_EVENTS(func)	用来处理所有的鼠标事件.

处理按钮和鼠标指针移动事件

按钮和指针移动事件是你想要处理的最主要的鼠标事件。

要检测当产生某个事件时状态键的状态，可以使用AltDown,MetaDown,ControlDown或者ShiftDown等函数.使用CmdDown函数来检测Mac OS X平台上的Meta键或者别的平台上的Control键的状态.本章稍后的"状态键变量"小节会对这些函数进行更详细的介绍.

要检测那个鼠标按钮正被按下,你可以使用LeftIsDown, MiddleIsDown和RightIsDown函数,或者你可以使用wxMOUSE_BTN_LEFT, wxMOUSE_BTN_MIDDLE, wxMOUSE_BTN_RIGHT或wxMOUSE_BTN_ANY参数来调用Button函数.要注意这些函数通常只是反应在事件产生那个时刻鼠标的状态,而不是反应鼠标的状态改变.(译者注:换句话说,两续同样按钮的两个事件中的按钮状态可能是一样的).

在Mac OS X上,Command键被翻译成Meta,Option键是Alt.因为在Mac系统上通常使用的是一键鼠标,当用户按下Control键点击鼠标的时候将产生右键单击事件.因此在MacOS上没有按下Control键时进行右键单击这样的事件,除非你正在使用的是一个两键或者三键的鼠标.

你还可以用下面的函数来或者鼠标事件的类型:Dragging (某个键正按下时鼠标移动), Moving (鼠标正在移动而没有鼠标键被按下), Entering, Leaving, ButtonDown, ButtonUp, ButtonDClick, LeftClick, LeftDClick, LeftUp, RightClick, RightDClick, RightUp, ButtonUp和IsButton等.

你可以使用GetPosition函数或者GetX和GetY函数获取鼠标指针当前的设备单位位置,也可以给GetLogicalPosition函数传递某个设备上下文参数以便得到对应的逻辑位置.

下面的例子演示了一个涂鸦程序中的鼠标处理过程:

```
BEGIN_EVENT_TABLE(DoodleCanvas, wxWindow)
    EVT_MOUSE_EVENTS(DoodleCanvas::OnMouseEvent)
END_EVENT_TABLE()
void DoodleCanvas::OnMouseEvent(wxMouseEvent& event)
{
    static DoodleSegment *s_currentSegment = NULL;
    wxPoint pt(event.GetPosition());
    if (s_currentSegment && event.LeftUp())
    {
        // 鼠标按钮释放的时候停止当前线段
        if (s_currentSegment->GetLines().GetCount() == 0)
        {
            // 释放线段记录并且释放指针
            delete s_currentSegment;
            s_currentSegment = (DoodleSegment *) NULL;
        }
        else
        {
            // 已经得到一个有效的线段,把它存下来
            DrawingDocument *doc = GetDocument();
            doc->GetCommandProcessor()->Submit(
            new DrawingCommand(wxT("Add Segment"), DOODLE_ADD,
                                    doc, s_currentSegment));
            doc->Modify(true);
            s_currentSegment = NULL;
        }
    }
    else if (m_lastX &gt; -1 && m_lastY &gt; -1 && event.Dragging())
    {
        //正在拖动鼠标,增加一行到当前的线段中
        if (!s_currentSegment)
            s_currentSegment = new DoodleSegment;
        DoodleLine *newLine = new DoodleLine(m_lastX, m_lastY, pt.x, pt.y);
        s_currentSegment->GetLines().Append(newLine);
        wxClientDC dc(this);
        DoPrepareDC(dc);
        dc.SetPen(*wxBLACK_PEN);
        dc.DrawLine( m_lastX, m_lastY, pt.x, pt.y);
    }
    m_lastX = pt.x;
    m_lastY = pt.y;
}
```

在上面的应用程序中,线段被存在文档类型.当用户使用鼠标左键在窗口上拖拽时,上面的函数增加一个线条到当前的线段中,并且把它画出来, 当用户释放左键的时候,当前的线段被提交到文档类进行处理(文档类是wxWidgets的文档视图框架的一部分) ,以便进一步实现文档的重做或

者撤消动作,而在窗口的OnPaint函数(代码没有被展示) 中,整个文档被重绘.在第19章"使用文档和视图"中,我们会完整的介绍这个例子.

如果想让这个程序更专业一点,可以在鼠标按下的时候捕获鼠标并且在鼠标释放的时候释放捕获,以便当鼠标左键按下并且移出窗口的时候仍然可以收到鼠标事件.

处理鼠标滚轮事件

当处理鼠标滚轮事件的时候,你可以使用GetWheelRotation函数获得滚轮滚过的位置的大小(可能为负数).用这个数除以 GetWheelDelta以便得到实际滚动行数的值.多数的设备每个GetWheelDelta发送一个滚轮事件,但是将来的设备也许会以更快的频率发送事件,因此你需要进行这种计算以便只有在滚轮滚过一整行的时候才滚动窗口,或者如果你可以滚动半行也可以.你还要把用户在控制面板中设置的滚轮每次滚动数量计算进去,这个数目可以通过GetLinesPerAction函数获得,要乘以这个值来得到实际用户希望滚动的数量.

另外,鼠标滚轮还可以被设置为每次滚动一页,你需要调用IsPageScroll函数来判断是否属于这种情况.

我们来举个例子,下面的代码是wxScrolledWindow的默认滚轮处理事件处理函数,其中的变量m_wheelRotation对已经滚动的位置进行累加,只有在滚动超过一行的时候才进行滚动.

```
void wxScrollHelper::HandleOnMouseWheel(wxMouseEvent& event)
{
    m_wheelRotation += event.GetWheelRotation();
    int lines = m_wheelRotation / event.GetWheelDelta();
    m_wheelRotation -= lines * event.GetWheelDelta();
    if (lines != 0)
    {
        wxScrollWinEvent newEvent;
        newEvent.SetPosition(0);
        newEvent.SetOrientation(wxVERTICAL);
        newEvent.m_eventObject = m_win;
        if (event.IsPageScroll())
        {
            if (lines &gt; 0)
                newEvent.m_eventType = wxEVT_SCROLLWIN_PAGEUP;
            else
                newEvent.m_eventType = wxEVT_SCROLLWIN_PAGEDOWN;
            m_win->GetEventHandler()->ProcessEvent(newEvent);
        }
        else
        {
            lines *= event.GetLinesPerAction();
            if (lines &gt; 0)
                newEvent.m_eventType = wxEVT_SCROLLWIN_LINEUP;
            else
                newEvent.m_eventType = wxEVT_SCROLLWIN_LINEDOWN;
            int times = abs(lines);
            for (; times &gt; 0; times)
                m_win->GetEventHandler()->ProcessEvent(newEvent);
        }
    }
}
```

6.2 处理键盘事件

键盘事件是由wxKeyEvent类表示的.总共有三种不同类型的键盘事件,分别为:键按下,键释放和字符事件.键按下和键释放事件是原始事件,而字符事件是翻译事件,我们马上会描述字符事件,不过在这之前先要清楚,如果一个按键被长时间按下,你通常就收到很多个键按下事件,而只收到一个键释放事件,,因此不要以为一个键释放事件一定对应一个键按下事件,这种想法是错误的.

要想接收到键盘事件,你的窗口必须拥有键盘焦点,这可以通过函数wxWindow::SetFocus来设置,比如当鼠标点击窗口的时候可以调用这个函数.

下表列出了对应的事件映射宏

EVT_KEY_DOWN(func)	用来处理**wxEVT_KEY_DOWN**事件 **(**原始按键按下事件**).**
EVT_KEY_UP(func)	用来处理wxEVT_KEY_UP事件 (原始的按键释放).
EVT_CHAR(func)	用来处理wxEVT_CHAR事件 (已经翻译的按键按下事件).

接下来描述一下你可以在事件处理函数中使用的处理函数.

要获得按键编码,你可以使用GetKeyCode函数(在Unicode版本中,你还可以使用GetUnicodeKeyCode函数).下表列出了所有的按键编码:

WXK_BACK	**WXK_RIGHT**
WXK_TAB	WXK_DOWN
WXK_RETURN	WXK_SELECT
WXK_ESCAPE	WXK_PRINT
WXK_SPACE	WXK_EXECUTE
WXK_DELETE	WXK_SNAPSHOT
WXK_INSERT	WXK_START
WXK_HELP	WXK_LBUTTON
WXK_RBUTTON	WXK_NUMPAD0
WXK_CANCEL	WXK_NUMPAD1
WXK_MBUTTON	WXK_NUMPAD2
WXK_CLEAR	WXK_NUMPAD3
WXK_SHIFT	WXK_NUMPAD4
WXK_CONTROL	WXK_NUMPAD5
WXK_MENU	WXK_NUMPAD6

WXK_PAUSE	WXK_NUMPAD7
WXK_CAPITAL	WXK_NUMPAD8
WXK_PRIOR	WXK_NUMPAD9
WXK_NEXT	WXK_END
WXK_MULTIPLY	WXK_HOME
WXK_ADD	WXK_LEFT
WXK_SEPARATOR	WXK_UP
WXK_SUBTRACT	WXK_DECIMAL
WXK_PAGEDOWN	WXK_DIVIDE
WXK_NUMPAD_SPACE	WXK_F1
WXK_NUMPAD_TAB	WXK_F2
WXK_NUMPAD_ENTER	WXK_F3 WXK_F4
WXK_NUMPAD_F1	WXK_F5
WXK_NUMPAD_F2	WXK_F6
WXK_NUMPAD_F3	WXK_F7
WXK_NUMPAD_F4	WXK_F8
WXK_NUMPAD_HOME	WXK_F9
WXK_NUMPAD_LEFT	WXK_F10
WXK_NUMPAD_UP	WXK_F11
WXK_NUMPAD_RIGHT	WXK_F12
WXK_NUMPAD_DOWN	WXK_F13
WXK_NUMPAD_PRIOR	WXK_F14
WXK_NUMPAD_PAGEUP	WXK_F15
WXK_NUMPAD_NEXT	WXK_F16
WXK_NUMPAD_PAGEDOWN	WXK_F17
WXK_NUMPAD_END	WXK_F18
WXK_NUMPAD_BEGIN	WXK_F19
WXK_NUMPAD_INSERT	WXK_F20
WXK_NUMPAD_DELETE	WXK_F21
WXK_NUMPAD_EQUAL	WXK_F22
WXK_NUMPAD_MULTIPLY	WXK_F23
WXK_NUMPAD_ADD	WXK_F24

WXK_NUMPAD_SEPARATOR	WXK_NUMPAD_SUBTRACT
WXK_NUMLOCK	WXK_NUMPAD_DECIMAL
WXK_SCROLL	WXK_NUMPAD_DIVIDE
WXK_PAGEUP	

要判断在按键的时候是否有状态键按下,可以使用AltDown, MetaDown, ControlDown或者ShiftDown函数. HasModifiers函数在有Control或者Alt键按下的时候返回True(不包括Shift和Meta键).

你可以使用CmdDown函数来代替ControlDown或者MetaDown函数,它在Mac OSX上调用MetaDown而在别的平台上调用ControlDown.在接下来的部分会对此进行解释.

GetPosition函数返回按键的时候的鼠标指针相对于窗口客户区原点的位置.

提示:如果你在键盘事件处理函数中没有调用event.Skip()函数,对应的字符事件将不会产生.在某些平台上,全局的快捷键也会不起作用.

字符事件处理的例子

下面列出的代码是随书光盘examples/chap12/thumbnail目录中wxThumbnailCtrl类的事件处理函数:

```
BEGIN_EVENT_TABLE( wxThumbnailCtrl, wxScrolledWindow )
    EVT_CHAR(wxThumbnailCtrl::OnChar)
END_EVENT_TABLE()
void wxThumbnailCtrl::OnChar(wxKeyEvent& event)
{
    int flags = 0;
    if (event.ControlDown())
        flags |= wxTHUMBNAIL_CTRL_DOWN;
    if (event.ShiftDown())
        flags |= wxTHUMBNAIL_SHIFT_DOWN;
    if (event.AltDown())
        flags |= wxTHUMBNAIL_ALT_DOWN;
    if (event.GetKeyCode() == WXK_LEFT ||
        event.GetKeyCode() == WXK_RIGHT ||
        event.GetKeyCode() == WXK_UP ||
        event.GetKeyCode() == WXK_DOWN ||
        event.GetKeyCode() == WXK_HOME ||
        event.GetKeyCode() == WXK_PAGEUP ||
        event.GetKeyCode() == WXK_PAGEDOWN ||
        event.GetKeyCode() == WXK_PRIOR ||
        event.GetKeyCode() == WXK_NEXT ||
        event.GetKeyCode() == WXK_END)
    {
        Navigate(event.GetKeyCode(), flags);
    }
    else if (event.GetKeyCode() == WXK_RETURN)
    {
        wxThumbnailEvent cmdEvent(
            wxEVT_COMMAND_THUMBNAIL_RETURN,
            GetId());
        cmdEvent.SetEventObject(this);
        cmdEvent.SetFlags(flags);
        GetEventHandler()->ProcessEvent(cmdEvent);
    }
    else
        event.Skip();
}
```

为了代码更简洁,方向键处理时候调用了另外一个单独的函数Navigate,而回车键则产生了一个更高一级的事件,这个事件可以被使用这个类的应用程序捕获并且处理,对于所有其它的按键,调用Skip函数以便应用程序的其它部分可以继续处理.

按键编码翻译

键盘事件提供的是未翻译的按键编码,而字符事件提供的是翻译以后的字符编码,对于未翻译的按键编码来说,子母永远是大些字符,其它字符则是在WXK_XXX中定义的字符.而对于已经翻译的按键编码来说,字符的值和同样的按键在一个文本编辑框中被按下以后在编辑框中产生的字符相同.

举个简单的例子,当一个单独的A键按下的时候,在KEY_DOWN事件中的字符编码是大字子母A的ASCII码65,而在相应的字符事件中的字符编码是小写的ASCII的a,编码为97.换句话说,当Shift和A键同时被按下时,上述两个事件中的编码是一样的,都是大写的A(65).

从这个小例子中我们可以清晰的看到,我们可以从键盘按下事件中的键盘编码和Shift键状态计算出相应的ASCII码,但是通常来说,如果你希望处理的是 ASCII码,你应该使用字符事件EVT_CHAR,因为对于非子母按键来说,如何翻译是和键盘布局有关的,只有系统本身才能对按键事件进行很好的翻译.

另外一种自动完成的按键翻译是那种带有Control键的翻译:比如说Ctrl+A,在KeyDown事件中,字符编码仍然是A,但是在字符事件中则为ASCII的1,因为ASCII中定义这个组合键的编码为1.

如果你对在你的系统中这种系统相关的键盘行为感兴趣,可以编译和运行键盘例子程序(在samples/keyboard目录中)然后按每个键试一下.

修饰键变量

在windows平台上,有Control和Alt两个修饰键,那个特殊的window键表现Meta键的行为.在Unix平台上,表现Meta键的按键是可以配置的(通过运行xmodmap来查看和改变现有配置).有时Numlock键也会被配置成Meta键,这是为什么在Numlock键被按下时,按下Meta键再按下别的键的时候,HasModifiers却返回False的原因.

在Mac OSX平台上,Command键(上面有一个苹果的标识)被翻译成Meta键,而Option键被翻译成Alt键.

各个平台上修饰键的不同如下表所示,其中wxWidgets采用的修饰键名放在第一栏,三个主要平台上对应的键放在后面三栏.

Modifier	Key on Windows	Key on Unix	Key on Mac	
Shift	Shift	Shift	Shift	
Control	Control	Control	Control	
Alt	Alt	Alt	Option	
Meta	Windows	(Configurable)	Command	

因为在Mac OSX上使用Command键而在别的平台上使用Control键,你可以使用wxKeyEvent的CmdDown函数来判断这个键在不同的平台上是否被按下.

另外除了在键盘事件处理函数中判断一个修饰键是否被按下以外,你还可以使用wxGetKeyState函数加上对应的键盘编码作为参数来判断某个键是否被按下,

加速键

加速键是为了实现通过某种组合键来快速执行菜单命令.加速键的处理是在所有的键盘事件(包括字符事件)之后.标准的加速键包括Ctrl+ O用来打开一个文件,Ctrl+V用来把剪贴板上的数据粘贴到应用程序中等.最简单的定义加速键的方法是在菜单项定义函数中使用下面的代码:

```
menu->Append(wxID_COPY, wxT("Copy\tCtrl+C"));
```

wxWidgets把"\t"后面的内容翻译为加速键增加到菜单的加速键表中.在上面的例子中,用户按Ctrl+C组合键的效果和用户选择这个菜单的效果是完全一样的.

你可以使用Ctrl,Alt和Shift以及它们的各种组合,然后加一个+号或者-号再跟一个字符或者功能键,比如下面的这些加速键都是合法的加速键: Ctrl+B, G, Shift-Alt-K, F9, Ctrl+F3, Esc 和Del. 在你的加速键定义中可以使用下面的名字: Del, Back, Ins, Insert, Enter, Return, PgUp, PgDn, Left, Right, Up, Down, Home, End, Space, Tab, Esc和 Escape. 这些命令是大小写无关的(你想怎样使用大小写都可以).

注意在Mac OSX平台上,一个定义为Ctrl的加速键实际上代表的是Command键.

另外一种设置加速键的方法是使用wxAcceleratorEntry对象定义一个加速键表,然后使用wxWindow:: SetAcceleratorTable函数将其和某个窗口绑定.每一个wxAcceleratorEntry的记录是由一个修饰键比特位值和一个字符或者功能键以及一个窗口标识符组成的,如下所示:

```
wxAcceleratorEntry entries[4];
entries[0].Set(wxACCEL_CTRL,  (int) 'N',     wxID_NEW);
entries[1].Set(wxACCEL_CTRL,  (int) 'X',     wxID_EXIT);
entries[2].Set(wxACCEL_SHIFT, (int) 'A',     wxID_ABOUT);
entries[3].Set(wxACCEL_NORMAL, WXK_DELETE,   wxID_CUT);
wxAcceleratorTable accel(4, entries);
frame->SetAcceleratorTable(accel);
```

你可以同时使用多个加速键表,也可以混合使用菜单项加速键和加速键表,如果你想给一个菜单项指定多个加速键,这将是非常有用的,因为你不可能在一个菜单项中指定多个加速键.

6.3 处理游戏手柄事件

wxJoystick类让你可以在windows平台或者linux平台上使用一到两个游戏手柄.典型的使用方法是,你创建wxJoystick的一个实例,并且传递给它wxJOYSTICK1或者wxJOYSTICK2的参数,并且以全局指针保持这个实例.当你需要处理手柄事件的时候,使用 SetCapture函数和一个窗口指针作为参数,来使的这个窗口收到游戏手柄事件,当你不需要使用手柄的时候,调用ReleaseCapture函数移出手柄事件.当然,你完全可以在应用程序初始化的时候调用SetCapture函数而在应用程序退出的时候调用ReleaseCapture函数,以便在整个应用程序生命周期内处理游戏手柄事件.

在开始描述详细的函数和事件之前,让我们先看一下wxWidgets的发行版中samples/joystick目录中的例子.在这个例子中,用户可以使用游戏手柄上的某个按钮画线,并且在按下按钮的时候播放一个声音片断.

下面大概列出了应用程序的起始代码,首先,应用程序通过创建一个临时的游戏手柄对象来检查系统是否有安装手柄,如果没有则退出应用程序.然后装载声音文件,并且获取手柄的最大活动范围以便在窗口上画画的时候实现合理的缩放使得画画的区域充满整个绘画窗口.

```
#include "wx/wx.h"
#include "wx/sound.h"
#include "wx/joystick.h"
bool MyApp::OnInit()
{
    wxJoystick stick(wxJOYSTICK1);
    if (!stick.IsOk())
    {
        wxMessageBox(wxT("No joystick detected!"));
        return false;
    }
    m_fire.Create(wxT("buttonpress.wav"));
    m_minX = stick.GetXMin();
    m_minY = stick.GetYMin();
    m_maxX = stick.GetXMax();
    m_maxY = stick.GetYMax();
    // Create the main frame window
    ...
    return true;
}
```

MyCanvas是那个用来接收和处理手柄事件的窗口,下面的MyCanvas类的实现部分:

```
BEGIN_EVENT_TABLE(MyCanvas, wxScrolledWindow)
    EVT_JOYSTICK_EVENTS(MyCanvas::OnJoystickEvent)
END_EVENT_TABLE()
MyCanvas::MyCanvas(wxWindow *parent, const wxPoint& pos,
    const wxSize& size):
    wxScrolledWindow(parent, wxID_ANY, pos, size, wxSUNKEN_BORDER)
{
    m_stick = new wxJoystick(wxJOYSTICK1);
    m_stick->SetCapture(this, 10);
}
MyCanvas::~MyCanvas()
{
    m_stick->ReleaseCapture();
    delete m_stick;
}
void MyCanvas::OnJoystickEvent(wxJoystickEvent& event)
{
    static long xpos = -1;
    static long ypos = -1;
    wxClientDC dc(this);
    wxPoint pt(event.GetPosition());
    // if negative positions are possible then shift everything up
    int xmin = wxGetApp().m_minX;
    int xmax = wxGetApp().m_maxX;
    int ymin = wxGetApp().m_minY;
    int ymax = wxGetApp().m_maxY;
    if (xmin &lt; 0) {
        xmax += abs(xmin);
        pt.x += abs(xmin);
    }
    if (ymin &lt; 0) {
        ymax += abs(ymin);
        pt.y += abs(ymin);
    }
    // Scale to canvas size
    int cw, ch;
    GetSize(&cw, &ch);
    pt.x = (long) (((double)pt.x/(double)xmax) * cw);
    pt.y = (long) (((double)pt.y/(double)ymax) * ch);
    if (xpos &gt; -1 && ypos &gt; -1 && event.IsMove() && event.ButtonIsDown())
    {
        dc.SetPen(*wxBLACK_PEN);
        dc.DrawLine(xpos, ypos, pt.x, pt.y);
    }
    xpos = pt.x;
    ypos = pt.y;
    wxString buf;
    if (event.ButtonDown())
        buf.Printf(wxT("Joystick (%d, %d) Fire!"), pt.x, pt.y);
    else
        buf.Printf(wxT("Joystick (%d, %d)"), pt.x, pt.y);
    frame->SetStatusText(buf);
    if (event.ButtonDown() && wxGetApp().m_fire.IsOk())
    {
        wxGetApp().m_fire.Play();
    }
}
```

wxJoystick的事件

wxJoystick类产生wxJoystickEvent类型的事件,下表列出了所有相关的事件映射宏:

EVT_JOY_BUTTON(func)	用来处理**wxEVT_JOY_BUTTON_DOWN**事件,手柄上的某个按钮被按下的时候产生**.**
EVT_JOY_BUTTON(func)	用来处理wxEVT_JOY_BUTTON_UP事件,某个按钮被释放的时候产生.
EVT_JOY_MOVE(func)	用来处理wxEVT_JOY_MOVE事件,手柄在X-Y平面上产生移动的时候产生.
EVT_JOY_ZMOVE(func)	用来处理wxEVT_JOY_ZMOVE事件,手柄在Z方向上产生移动的时候产生.
EVT_JOYSTICK_EVENTS(func)	处理所有的手柄事件.

wxJoystickEvent的成员函数

下面列出了所有wxJoystickEvent的成员函数,你可以使用它们获取关于手柄事件更详细的信息,当然,首先还是GetEventType函数,这个函数在你使用EVT_JOYSTICK_EVENTS宏的时候告诉你你正在处理哪种类型的手柄事件.

调用ButtonDown函数来检测是否是按钮被按下事件,你可以传递可选的wxJOY_BUTTONn(n代表1,2,3或4)来测试是否某个特定的按钮正被按下,或者使用wxJOY_BUTTON_ANY如果你不关心具体是哪个按钮被按下的话.ButtonUp则用相似的方法来检测是按钮被释放事件.而IsButton则相当于调用ButtonDown() || ButtonUp().

要判断是否某个按钮被按下的状态而不是事件本身,你可以使用ButtonIsDown函数,它和ButtonDown的参数相似.或者你可以直接使用GetButtonState函数和类似的参数来返回指定按钮的按下状态的比特位.

使用IsMove函数来判断是否是一个XY平面的移动事件,使用IsZMove判断是否是一个Z平面的移动事件.

GetPosition返回游戏手柄当前XY平面的座标,而GetZPosition返回Z方向的深度值(当然,如果手柄支持的话).

最后,你可以使用GetJoystick来判断这个事件是哪个手柄(wxJOYSTICK1还是wxJOYSTICK2)产生的.

wxJoystick成员函数

我们没有列出游戏手柄类所有的成员函数,你可以参考wxWidgets的手册,不过下面是其中最有趣的几个:

正象我们在前面的例子中看到的那样,SetCapture函数需要被调用如果你想让某个窗口处理手柄事件,而一个匹配的 ReleaseCapture函数调用用来释放这次捕获,以便允许别的应用程序使用手柄.为避免别的程序正在使用手柄,或者手柄不能正常工作,你应该在调用SetCapture之前先调用IsOK函数来判断手柄的状态.你还可以通过这些函数来获取手柄的能力集:GetNumberButtons, GetNumberJoysticks,GetNumberAxes,HasRudder等等.

GetPosition函数和GetButtonState函数可以让你在手柄事件处理函数以外的地方获取手柄的状态.

你的应用程序还可能需要调用GetXMin,GetXMax等这些类似的函数来判断一个手柄支持的范围.

第六章小结

在这一章里,我们介绍了怎样处理来自鼠标,键盘以及游戏手柄的输入事件,现在你可以给你的应用程序增加复杂的交互性代码了.你可以参考wxWidgets 自带的例子比如: samples/keyboard, samples/joytest和 samples/dragimag,也可以参考光盘上的examples/chap12目录里的wxThumbnailCtrl类来了解更具体的信息.

在接下来的一章里,我们会介绍一下怎样对窗口里的控件进行可变大小的,轻便的,可以友善转换的布局.之所以能支持这些诱人的特性,是因为我们有强大而灵活的布局控件.

第七章使用布局控件进行窗口布局

所有的图形界面设计者都会告诉你,人们对于可以看见的对象的排列是非常敏感的.一个好的GUI设计框架一定要允许创建具有吸引力的窗口布局.但是和打印布局不同,应用程序的窗口可能由于大小,字体设置甚至是语言的不同进行不同的变化.对于一个平台无关的设计来说,还要考虑同一个控件在不同的平台上可能有不同的外观和尺寸.这意味着使用绝对大小和位置来进行窗口布局几乎是行不通的.本章描述的wxWidgets的布局控件,让你可以灵活的进行非常复杂的窗口布局.如果这听上去有一点让人畏惧,那么记住有一些工具可以帮助你以图形化的方式使用布局控件创建布局,比如包含在附赠光盘中的DialogBlocks 工具,使用它们你几乎用不着手动设置任何布局.

7.1 窗口布局基础

在深入介绍窗口布局之前,我们先来大概了解一下什么时候你需要进行窗口布局以及你拥有的几种选择.

一个最简单的情况是你拥有一个frame窗口,其中只有一个客户窗口位于这个frame的客户区.这是最简单的情况,wxWidgets将为你完成所有的布局工作,将那个客户区窗口缩放到刚好适合frame的客户区的大小.wxWidgets也会管理frame的菜单条,工具条和状态条(如果有的话).当然,如果你想使用两个工具条,你至少要管理其中的一条,而如果frame窗口的客户区拥有超过一个窗口,你将不得不自己单独管理它们所有的大小和位置,比如你可能需要在OnSize事件中计算每一个窗口的大小并且设置它们的新位置.当然,你也可以使用窗口布局控件.类似的,如果你创建了一个定制的控件,这个控件拥有多个子窗口,你也需要安排这些子窗口的布局以便当你的这个控件被别人使用而且默认大小改变的时候,对那些子窗口进行合理的布局.

另外,大多数应用程序都需要创建自己的对话框,有时候有些程序会创建很多个定制的对话框,这些对话框可能被改变大小,在这种情况下,对话框上所有的控件的大小也应该发生相应的改变,以便即使这个对话框已经比最初设计的时候大了很多,这些控件看上去也不会显得很奇怪.另外应用程序的语言也可能改变,某些在默认语言中很短的标签,在另外一种语言中可能会变得很长.如果你的应用程序要应付成百上千中这种对话框,相信即使使用窗口布局控件,维护它们也是一个令人望而生畏的工作,还好幸运的是我们还可以使用一些工具让所有这些事情变得不那么恐怖甚至还有一点点的乐趣在其中.

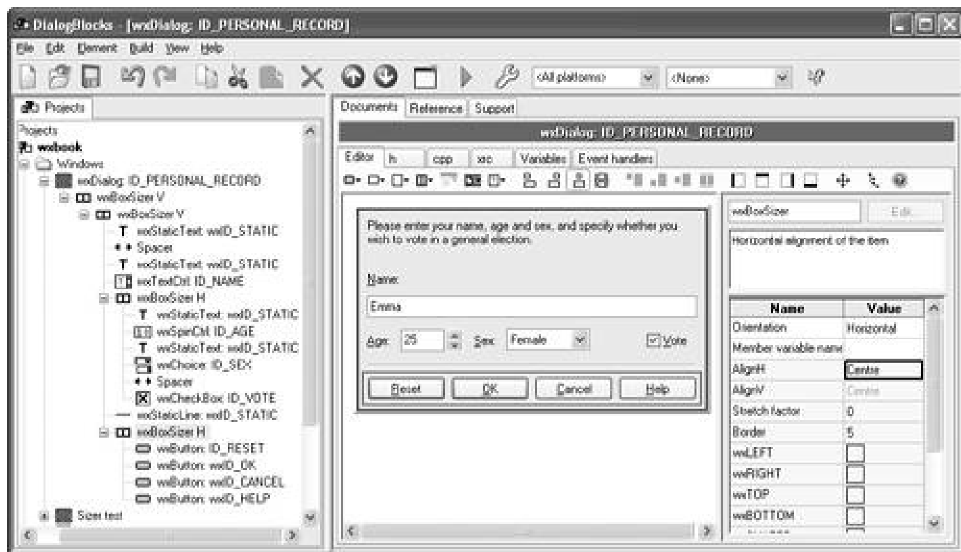
如果你选择使用布局控件,你需要自己决定怎样创建和发布它们.你可以自己写或者使用工具来创建C++或者其它语言的代码,或者你可以直接使用XRC文件,XRC文件用来将布局的定义保存在一个Xml文件中,可以被应用程序动态加载,也可以通过wxrc工具将其编译成C++源文件以便和别的源文件编译成一个单独的可执行文件.大多数的wxWidgets对话框编辑器都既支持产生C++的代码,又支持生成XRC文件.怎样作决定完全取决于你自己的审美观,有些人喜欢把一切都放在一个C++代码中以保持足够的灵活性,而另外一些人则喜欢把实现功能的代码和产生界面的代码分开.

接下来的小节用来描述布局控件的原理,再往后一个小节则用来描述怎样使用各种具体的布局控件来编程.

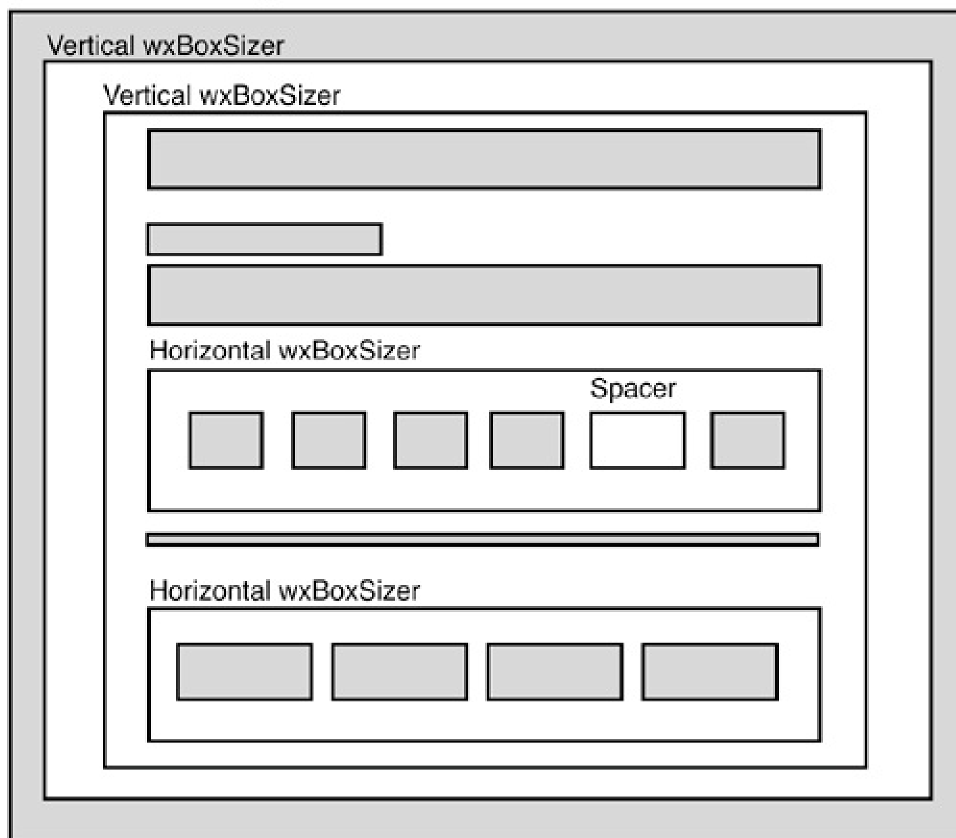
7.2 窗口布局控件

wxWidgets使用的窗口布局控件的算法和其它GUI程序开发框架的算法,例如Java的AWT,GTK+以及Qt等是非常相似的.它们都是基于这样的一个假设,那就是每一个窗口可以报告它们自己需要的最小尺寸以及当它们父窗口的大小发生改变的时候它们的可伸缩能力.通常这也意味着程序代码中没有给对话框中的控件设定固定的大小,取而代之的是设置了一个窗口布局控件,这个窗口布局控件会被寻问它最需要的合适大小,而窗口布局控件则会一次询问它内部的那些窗口,空白区域,控件以及其它的窗口布局控件最合适的大小,以此类推.要注意布局控件并非是wxWindow的派生类,因此不具有TAB顺序,所需要的系统开销也比一个真实的窗口要小的多.布局控件建立的是一种包含继承关系,在一个复杂的对话框里,这种包含继承关系的层级可能会很深,但是所有这些布局控件中的窗口或控件在窗口继承关系中可能都是兄弟控件,它们都以这个对话框作为自己的父窗口.

在对话框编辑软件中,布局控件的包含继承关系以一种更直观的图形化方式表示.下图演示了Anthemion软件公司的 DialogBlocks软件编辑一个个人对话框时候的样子,这个对话框我们会在第9章,"创建自己的对话框"中作为一个例子.一个红色的方框围绕在当前选择的控件上,而蓝色的方框则用来指示其直接父容器的范围,左边显示的是对话框的容器继承关系树,当然所有这些控件还是有它们自己的窗口继承关系的,正如我们前面提到的那样,窗口继承关系树和容器继承关系树有很大的区别.



为了更清楚的说明容器继承关系和窗口继承关系的不同,我们用下图来大概的表示上图中的容器继承关系.下图中,阴影部分代表实际的窗口,而白色区域则代表布局控件.可以看出,对话框首先使用了两个垂直布局控件,以便在对话框周围释放出一个合理的边界区域,里面的那个垂直布局控件中有两个水平布局控件和一些其它的控件,其中一个水平布局控件中还定义了一截空白区域以便使得其中一个控件远离同组中另外的控件.正象你看到的那样,使用布局控件就象使用一堆大小不等的硬纸片进行摆放,然后把各个窗口控件放置到硬纸片的合适的位置.当然这个比喻并不完全贴切因为,硬纸片的大小是不会伸缩的.



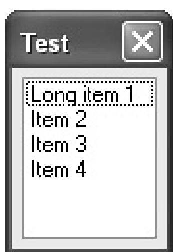
目前为止wxWidgets总共支持五类布局控件,每一中布局控件或者用来实现一种特殊的布局方式,或者用来实现和布局相关的一种特殊的功能比如在某些控件周围围绕一个静态的文本框.接下来的小节我们会对它们进行一一的介绍.

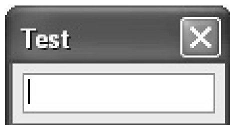
布局控件的通用特性

所有的布局控件都是容器,这就是说,它们都是用来容纳一个或多个别的窗口或者元素的,不论每个单独的布局控件怎样排放它们的子元素,所有的子元素都必然有下面这些通用的特性.

最小大小: 布局控件中的每个元素都有计算自己的最小大小的能力(这往往是通过每个元素的DoGetBestSize函数计算出来的).这是这个元素的自然大小.举例来说,一个复选框的自然大小等于其复选框图形的大小加上其标签的最合适大小.当然,你可以给某个控件在其构造函数中指定固定的大小,并且在把它增加到布局控件中时指定wxFIXED_MINSIZE以改变自动计算的最小大小.需要注意的是,不是每个控件都可以计算自己的大小,对于类似列表框这样的控件来说,你必须清晰的指明它的大小,因为它们没有自然大小.另外一些控件则只拥有自然高度不拥有自然宽度,比如一个单行的文本框.下图演示了当对话框中只有一个控件的时候,以上三种控件怎样扩展对话框以适合自己的最小大小.







边界: 每个元素都应该有一个边界.所谓边界指的是用来和别的元素分开的空白区域,边界的最小大小必须被显式的指定,一般设置为5个象素.下图演示了对话框只有一个按钮控件但是指定了0,5和10作为最小边界值的时候的样子.



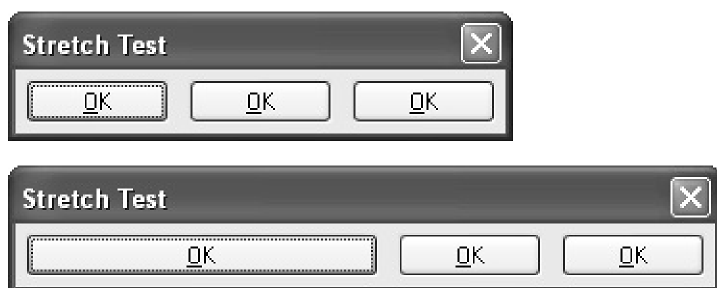




对齐方式: 每个元素都可以被以居中或者对齐某个边的方式放置.下图演示了一个水平的布局控件,在其中增加了一个列表框,一个和三个按钮,其中第一个按钮以居中方式增加,第二个则为上对齐,第三个则为下对齐方式.对齐既可以是水平方向的也可以是垂直方向的,但是对于大多数布局控件来说,只有一个方向是有效的.比如对于水平布局控件来说,只有垂直方向是有效的,因为水平方向的空间是被所有的子元素分割的,因此设置水平对齐方式是没有意义的(当然,为了达到水平对齐的效果,我们可能需要插入一个水平方向的空白区域,关于这点我们不作太多的说明).



伸缩因子: 如果一个布局控件的空间大于它所有子元素所需要的空间,那么我们需要一个机制来分割多余的空间.为了实现这个目的,布局控件中的每一个元素都可以指定一个伸缩因子,如果这个因子设置为默认值0,那么子元素将保持其原本的大小,大于0的值用来指定这个子元素可以分割的多余空间的比例,因此如果两个子元素的伸缩因子为1,其它子元素的伸缩机制为0,那么这两个子元素将会各占用多余空间的50%的大小,下图演示了一个对话框有三个按钮它们的初始大小和其中一个的伸缩因子设为1以后各自的大小.



注意在wxWidgets的手册中,有时不使用伸缩因子(stretch factor)这个词,而用比例(proportion)这个词表示相同的含义.

7.3 使用布局控件进行编程

现在我们开始使用布局控件进行窗口布局,首先,创建一个顶层的布局控件(任何类型的布局控件都可以),使用wxWindow::SetSizer函数将它和你的顶层窗口绑定.现在你可以在这个顶层布局控件中放置你的窗口或者其它控件元素了.如果你想让你的顶层窗口的大小适合所有控件所需要的大小,你可以调用wxSizer::Fit函数,将那个顶层窗口的指针作为其参数.想要顶层窗口在以后的执行过程中尺寸永远不小于初始尺寸,可以使用wxSizer:: SetSizeHints函数,将顶层窗口的指针作为参数,这将使得wxWindow::SetSizeHints函数以合适的参数被调用.

除了使用上面介绍的方法依次调用三个函数以外,你还可以直接通过调用wxWindow::SetSizerAndFit函数来达到同样的效果,这会使得上面的三个函数依次被调用.

如果在你的frame窗口里使用了panel,你可能不知道到底该给frame还是panel指定布局控件.这个问题应该这样看,如果你的frame窗口中只有一个panel,所有其它的窗口和控件都是panel的子窗口,那么wxWidgets已经知道怎样将这个panel以合适的大小和位置放置在frame上了,因此你只需要给panel绑定一个布局控件,以便其可以对所有panel的子窗口进行布局.而如果你的frame窗口中有多个panel,那么首先你不得不为frame绑定一个布局控件以便对panel进行布局,然后针对每个panel还应该绑定一个布局控件,以便对 panel中的子窗口进行布局.

接下来的小节里,我们来依次描述一下每一种布局控件类型以及使用它们的方法:

使用wxBoxSizer进行编程

wxBoxSizer可以将它的容器子元素进行横向或者纵向的排列(具体的排列方式在构造函数中指定).如果采用横向排列的方法,则子元素在纵向上可以指定居中,顶部对齐,底部对齐,如果采用纵向排列的方法,子元素在横向上可以指定居中,左对齐或者右对齐的方式.前一小节提到过的缩放因子用来指示在主要方向上的缩放,比如对于横向排列来说,缩放因子指的就是在横向上子元素的缩放比例. 下图演示了上一小节最后一幅图采用纵向排列的样子.



你可以使用wxBoxSizer的Add方法增加一个子元素:

```
// 增加一个窗口
void Add(wxWindow* window, int stretch = 0, int flags = 0,
         int border = 0);
// 增加一个布局控件
void Add(wxSizer* window, int stretch = 0, int flags = 0,
         int border = 0);
```

第一个参数是要增加的窗口或者布局控件的指针

第二个参数是前面说过的缩放因子

第三个参数是一个比特位列表,用来指示新增的子元素的对齐和边界的行为.对齐比特位用来指示当垂直排列的布局控件的宽度发生改变时子元素的水平对齐方式,或者是水平排列的布局控件的高度改变时子元素的垂直对齐方式,默认的值为 wxALIGN_LEFT | wxALIGN_TOP,可选的值列举在下表中:

0	子元素保留原始大小.
wxGROW	子元素随这布局控件一起改变大小. 等同于 wxEXPAND.
wxSHAPED	子元素保持原有比例按缩放因子缩放.
wxALIGN_LEFT	左对齐.
wxALIGN_RIGHT	右对齐.
wxALIGN_TOP	顶端对齐.
wxALIGN_BOTTOM	底部对齐.
wxALIGN_CENTER_HORIZONTAL	水平居中.
wxALIGN_CENTER_VERTICAL	垂直居中.
wxALIGN_CENTER	水平或者垂直居中. wxALIGN_CENTER_HORIZONTAL | wxALIGN_CENTER_VERTICAL.
wxLEFT	边界间隔位于子元素左面.
wxRIGHT	边界间隔位于子元素右面.
wxTOP	边界间隔位于子元素上面.
wxBOTTOM	边界间隔位于子元素下面.
wxALL	边界间隔位于子元素四周.wxLEFT | wxRIGHT | wxTOP | wxBOTTOM.

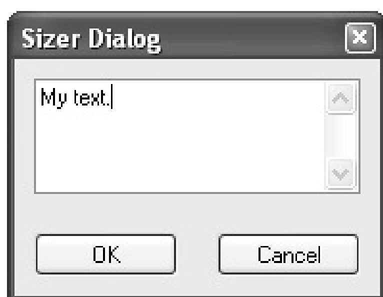
第四个参数指定边界间隔的大小

当然你也可以直接增加一段空白,下面演示了增加空白区域的几种方法:

```
// 增加一段空白 (旧方法)
void Add(int width, int height, int stretch = 0, int flags = 0,
         int border = 0);
// 增加一段固定大小的空白
void AddSpacer(int size);
// 增加一个可缩放的空白
void AddStretchSpacer(int stretch = 1);
```

上面第二种方法相当于调用Add(size, size, 0),第三种则相当于调用Add(0, 0, stretch).

我们来举这样一个例子,一个对话框包含一个多行文本框和两个位于底端的按钮.我们可以以这样的角度去看待这些窗口,首先是一个顶层的垂直布局,包含一个多行文本框和一个底层的子布局控件,这个子布局控件是一个水平的布局控件,它包含一个OK按钮,被放置在左面,和一个Cancel按钮,被放置在右面.当对话框的大小发生变化的时候,我们希望多行文本框随着对话框大小的变化而变化,而按钮则保持它们原来的大小,并且在水平方向上居中排列,如下图所示:



下面列出了实现上述对话框所使用的代码:

```
MyDialog::MyDialog(wxWindow *parent, wxWindowID id,
                     const wxString &title )
        :  wxDialog(parent, id, title,
                     wxDefaultPosition, wxDefaultSize,
                     wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
{
    wxBoxSizer  *topSizer = new wxBoxSizer( wxVERTICAL );
    // 创建一个最小大小为 100x60的多行文本框
    topSizer->Add(
        new wxTextCtrl( this, wxID_ANY, "My text.",
            wxDefaultPosition, wxSize(100,60), wxTE_MULTILINE),
        1,          // 垂直方向可缩放,缩放因子为1
        wxEXPAND|   // 水平方向可缩放
        wxALL,      // 四周都由空白边框
        10 );       // 空白边框大小为10
    wxBoxSizer *buttonSizer = new wxBoxSizer( wxHORIZONTAL );
    buttonSizer->Add(
       new wxButton( this, wxID_OK, "OK" ),
       0,           // 水平方向不可缩放
       wxALL,      // 四周有空白边框:(注意默认为顶部对齐)
       10 );       // 空白边框大小为10
    buttonSizer->Add(
       new wxButton( this, wxID_CANCEL, "Cancel" ),
       0,          // 水平方向不可缩放
       wxALL,      // 四周有空白边框:(注意默认为顶部对齐)
       10 );       // 空白边框大小为10
    topSizer->Add(
       buttonSizer,
       0,                // 垂直方向不可缩放
       wxALIGN_CENTER ); // 无边框并且居中对齐
    SetSizer( topSizer ); // 绑定对话框和布局控件
    topSizer->Fit( this );          // 调用对话框大小
    topSizer->SetSizeHints( this ); // 设置对话框最小大小
}
```

使用wxStaticBoxSizer编程

wxStaticBoxSizer是一个继承自wxBoxSizer的布局控件,除了实现wxBoxSizer的功能,另外还在整个布局的范围以外增加了一个静态的边框wxStaticBox,这个wxStaticBox需要手动创建并且在wxStaticBoxSizer的构造函数中作为参数传入,Add函数和wxBoxSizer的Add函数用法相同.

下图演示了使用wxStaticBoxSizer对一个复选框进行布局的样子:



对应的代码:

```
MyDialog::MyDialog(wxWindow *parent, wxWindowID id,
                   const wxString &title )
        : wxDialog(parent, id, title,
                   wxDefaultPosition, wxDefaultSize,
                   wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
{
    // 创建一个顶层布局控件
    wxBoxSizer* topLevel = new wxBoxSizer(wxVERTICAL);
    // 创建静态文本框和静态文本框布局控件
    wxStaticBox* staticBox = new wxStaticBox(this,
        wxID_ANY, wxT("General settings"));
    wxStaticBoxSizer* staticSizer = new wxStaticBoxSizer(staticBox,
        wxVERTICAL);
    topLevel->Add(staticSizer, 0,
        wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
    // 在其中增加一个复选框
    wxCheckBox* checkBox = new wxCheckBox( this, ID_CHECKBOX,
        wxT("&Show splash screen"), wxDefaultPosition, wxDefaultSize);
    staticSizer->Add(checkBox, 0, wxALIGN_LEFT |wxALL, 5);
    SetSizer(topLevel);
    topLevel->Fit(this);
    topLevel->SetSizeHints(this);
}
```

使用wxGridSizer编程

wxGridSizer布局控件可以以二维表的方式排列它的子元素,这个二维表的每个表格的大小都是相同的,都等于最长的那个表格的长度和最高的那个表格的高度.创建一个wxGridSizer需要指定它的行数和列数,以及一个额外的行间距和列间距.Add方法和wxBoxSizer的用法相同.

下图演示了一个两行三列的网格布局控件,由于有一个很大的按钮,导致这个格的大小很大,从而导致所有的表格的大小都跟着变大:



代码如下:

```
MyDialog::MyDialog(wxWindow *parent, wxWindowID id,
                   const wxString &title )
        : wxDialog(parent, id, title,
                   wxDefaultPosition, wxDefaultSize,
                   wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
{
    // 创建一个顶层网格布局控件
    wxGridSizer* gridSizer = new wxGridSizer(2, 3, 0, 0);
    SetSizer(gridSizer);
    wxButton* button1 = new wxButton(this, ID_BUTTON1, wxT("One"));
    gridSizer->Add(button1, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button2 = new wxButton(this, ID_BUTTON2, wxT("Two (the second button)"));
    gridSizer->Add(button2, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button3 = new wxButton(this, ID_BUTTON3, wxT("Three"));
    gridSizer->Add(button3, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button4 = new wxButton(this, ID_BUTTON4, wxT("Four"));
    gridSizer->Add(button4, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button5 = new wxButton(this, ID_BUTTON5, wxT("Five"));
    gridSizer->Add(button5, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button6 = new wxButton(this, ID_BUTTON6, wxT("Six"));
    gridSizer->Add(button6, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    gridSizer->Fit(this);
    gridSizer->SetSizeHints(this);
}
```

使用wxFlexGridSizer编程

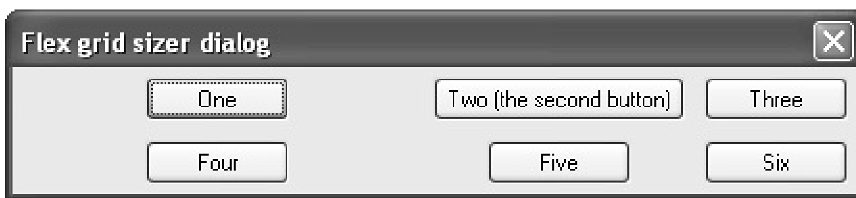
wxFlexGridSizer同样采用二维表来对其子元素进行布局,和wxGridSizer不同的是,它不要求所有的表格的大小都是一样的,只要求同一列上所有表格的宽度是相同的并且同一行上所有表格的高度是相同的,也就是说,行的高度或者列的宽度仅由这一行或者这一列上的子元素决定.另外还可以给行和列指定是否缩放,这意味着当整个布局控件的大小发生变化的时候,可以指定某些行或者列随着整个布局控件的缩放而缩放.

创建一个wxFlexGridSizer可以指定行数,列数额外的垂直间距和水平间距.调用Add函数的方法和wxBoxSizer相同.

下图演示了一个使用wxFlexGridSizer进行布局的对话框的样子,正如你看到的那样,和wxGridSizer相比整个布局显的更紧凑了,因为中间很宽的那一列不再影响其它列的宽度了.



初始情况下,我们看不出来设置第一列可以改变大小的效果,不过如果我们如下图所示的那样改变这个对话框的水平方向的大小,,我们就可以看到第一列占用了额外增加的空间,并且第一列的子元素也为居中方式显式.



代码如下:

```
MyDialog::MyDialog(wxWindow *parent, wxWindowID id,
                   const wxString &title )
        : wxDialog(parent, id, title,
                   wxDefaultPosition, wxDefaultSize,
                   wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
{
    //创建一个复杂网格布局控件
    wxFlexGridSizer* flexGridSizer = new wxFlexGridSizer(2, 3, 0, 0);
    this->SetSizer(flexGridSizer);
    //让第一列可变大小
    flexGridSizer->AddGrowableCol(0);
    wxButton* button1 = new wxButton(this, ID_BUTTON1, wxT("One"));
    flexGridSizer->Add(button1, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button2 = new wxButton(this, ID_BUTTON2, wxT("Two (the second button)"));
    flexGridSizer->Add(button2, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button3 = new wxButton(this, ID_BUTTON3, wxT("Three"));
    flexGridSizer->Add(button3, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button4 = new wxButton(this, ID_BUTTON4, wxT("Four"));
    flexGridSizer->Add(button4, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button5 = new wxButton(this, ID_BUTTON5, wxT("Five"));
    flexGridSizer->Add(button5, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxButton* button6 = new wxButton(this, ID_BUTTON6, wxT("Six"));
    flexGridSizer->Add(button6, 0, wxALIGN_CENTER_HORIZONTAL|
                               wxALIGN_CENTER_VERTICAL|wxALL, 5);
    flexGridSizer->Fit(this);
    flexGridSizer->SetSizeHints(this);
}
```

使用wxGridBagSizer编程

这种布局控件用来模拟现实世界中的那种固定位置和大小的基于布局控件的布局.它将它的子元素按照一个虚拟的网格进行排列,不过子元素的位置是通过wxGBPosition对象指定的,对象的大小使用wxGBSpan指定,对象的大小不仅限于一个网格.

创建wxGridBagSizer的可选参数包括垂直和水平方向的间隔(默认为0),Add函数需要提供的参数包括子元素的位置和大小,另外的可选标记和边框大小参数的意义和wxBoxSizer是一样的.

下图演示了一个使用wxGridBagSizer进行布局的例子,我们指定了其中一个按钮的大小为两个单元列,我们还指定了第二行和第三列的大小是可以变化的,这样当我们改变对话框的大小的时候,就会出现如下面另外一幅图的效果.



相关代码如下:

```
MyDialog::MyDialog(wxWindow *parent, wxWindowID id,
                   const wxString &title )
        : wxDialog(parent, id, title,
                   wxDefaultPosition, wxDefaultSize,
                   wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER)
{
    wxGridBagSizer* gridBagSizer = new wxGridBagSizer();
    SetTopSizer(gridBagSizer);
    wxButton* b1 = new wxButton(this, wxID_ANY, wxT("One (0,0)"));
    gridBagSizer->Add(b1, wxGBPosition(0, 0));
    wxButton* b2 = new wxButton(this, wxID_ANY, wxT("Two (2,2)"));
    gridBagSizer->Add(b2, wxGBPosition(2, 2), wxGBSpan(1, 2),
                      wxGROW);
    wxButton* b3 = new wxButton(this, wxID_ANY, wxT("Three (3,2)"));
    gridBagSizer->Add(b3, wxGBPosition(3, 2));
    wxButton* b4 = new wxButton(this, wxID_ANY, wxT("Four (3,3)"));
    gridBagSizer->Add(b4, wxGBPosition(3, 3));
    gridBagSizer->AddGrowableRow(3);
    gridBagSizer->AddGrowableCol(2);
    gridBagSizer->Fit(this);
    gridBagSizer->SetSizeHints(this);
}
```

7.4 更多关于布局的话题

这一节里,我们将讨论一些更深入的话题,在进行窗口布局的时候,你可以在脑子里考虑这些事情.

对话框单位

尽管布局控件可以让基本控件的大小随着平台的不同语言的不同进行相应的改变,但是有些情况下,你还是需要手动指定控件的大小(比如在对话框中增加一个列表框的时候).如果你希望这些手动指定的大小也随着平台的不同字体的不同进行相应的变化,你应该使用对话框单位来代替象素单位.对话框单位是基于应用程序当前字体的字符宽度和高度所取的一个平均值的,因此总能很好的和当前的字体对应.wxWidgets也提供了相关的转换函数包括: ConvertDialogToPixels,ConvertPixelsToDialog等,还包括一个宏wxDLG_UNIT(window, ptOrSz)用来直接将使用对话框单位wxPoint对象或者wxSize对象转换为象素单位.所以你可以使用下面的代码来指定那些你不得不指定的控件大小:

```
wxListBox* listBox = new wxListBox(parent, wxID_ANY,
    wxDefaultPosition, wxDLG_UNIT(parent, wxSize(60, 20)));
```

你也可以在XRC文件中使用对话框单位,只需要在相应的值前面增加一个"d"字符就可以了.

平台自适应布局

尽管不同平台的对话框的绝大部分都是相同的,但是在风格上确是存在着一些不同.比如在Windows和Linnx平台上,右对齐或者居中放置的OK,Cancel和Help按钮都是可以接受的,但是在Mac OsX上,Help按钮通常位于左面,而Cancel和OK按钮则通常依序位于右面.

要作到这种不同平台上按钮顺序的自适应,你需要使用wxStdDialogButtonSizer布局控件,这个控件继承自wxBoxSizer,因此使用方法并没有太大的不同,只是它依照平台的不同对某些按钮进行特殊的排列.

这个布局控件的构造函数没有参数,要增加按钮可以使用两种方法:传递按钮指针给AddButton函数,或者(日过你没有使用标准的标识符的话),使用SetAffirmativeButton, SetNegativeButton, and SetCancelButton来设置按钮的特性.如果使用AddButton,那么按钮应使用下面的这些标识符: wxID_OK, wxID_YES, wxID_CANCEL, wxID_NO, wxID_SAVE, wxID_APPLY, wxID_HELP和 wxID_CONTEXT_HELP.

然后,在所有的按钮都增加到布局控件以后,调用Realize函数以便布局控件调整按钮的顺序,如下面的代码所示:

```
wxBoxSizer* topSizer = new wxBoxSizer(wxVERTICAL);
dialog->SetSizer(topSizer);
wxButton* ok = new wxButton(dialog, wxID_OK);
wxButton* cancel = new wxButton(dialog, wxID_CANCEL);
wxButton* help = new wxButton(dialog, wxID_HELP);
wxStdDialogButtonSizer* buttonSizer = new wxStdDialogButtonSizer;
topSizer->Add(buttonSizer, 0, wxEXPAND|wxALL, 10);
buttonSizer->AddButton(ok);
buttonSizer->AddButton(cancel);
buttonSizer->AddButton(help);
buttonSizer->Realize();
```

或者作为一个更方便的手段,你可以使用wxDialog::CreateButtonSizer函数,它基于一些按钮标记的列表来自动创建平台自适应的按钮,并将其放在一个布局控件中,如果你查看src/generic目录中的对话框代码的实现,你会发现大量的地方使用了 CreateButtonSizer函数.这个函数支持的按钮标记如下表所示:

wxYES_NO	增加**YES**和**No**按钮各一个**.**
wxYES	增加一个标识符为wxID_YES的Yes按钮.
wxNO	增加一个标识符为wxID_NO的No按钮.
wxNO_DEFAULT	让No按钮作为默认按钮,否则Yes或OK按钮将成为默认按钮.
wxOK	增加一个标识符为wxID_OK的OK按钮.
wxCANCEL	增加一个标识符为wxID_CANCEL的Cancel按钮.
wxAPPLY	增加一个标识符为wxID_APPLY的Apply按钮.
wxHELP	增加一个标识符为wxID_HELP的Help按钮.

使用CreateButtonSizer函数,上面例子中的代码可以简化为:

```
wxBoxSizer* topSizer = new wxBoxSizer(wxVERTICAL);
dialog->SetSizer(topSizer);
topSizer->Add(CreateButtonSizer(wxOK|wxCANCEL|wxHELP), 0,
              wxEXPAND|wxALL, 10);
```

另外一种给不同的平台指定不同布局的方法是在XRC文件中指定平台属性.其中的参数部分的值可以通过一个"|"符号加上unix, win,mac或者os2来指定特定平台上的界面布局.在应用程序运行的时候,XRC文件将只会创建那些和当前运行平台符合的控件.另外如果没有使用 XRC的话,DialogBlocks程序还支持针对不同的平台生成预置条件的C++代码.

当然你也可以给不同的平台指定不同的XRC文件,不过这样作的话维护起来就有点不方便了.

动态布局

有时候你可能需要动态更改对话框的布局,比如你可以会增加一个"Detail"按钮,当这个按钮被按下的时候显式更多的选项,当然你可以使用平常的办法,调用wxWindow::Show函数来隐藏某个控件,不过wxSizer也提供了一个单独的方法,你可以使用wxSizer:: Show函数并且传递False参

数,以便告诉wxSizer不要计算其中的窗口的大小,当然调用这个函数以后,你需要调用wxSizer::Layout函数来强制更新对应的窗口.

第七章小结

布局控件可能需要花点时间才能慢慢习惯使用,因此,如果你觉得这一章的内容有点沉重,不要担心.掌握它们最好的办法是使用光盘中附带的 DialogBlocks软件进行各种各样的窗口布局,然后观看其自动产生的代码.你也可以参考wxWidgets自带的samples/layout中的例子.在你可以熟练使用它们以后,你会发现,它们在不同平台和不同语言下进行布局的能力,可以让你的产品极大的受益.

在下一章中,我们来看一看wxWidgets提供的标准对话框.

第八章使用标准对话框

本章描述了wxWidgets提供的标准的对话框,使用这些这些对话框,你只用很少的代码就可以用来显示一些信息或者从用户那里获取数据.熟悉这些标准的对话框将会节省你大量的时间,并且让你的程序显得更专业.wxWidgets在任何可能的时候使用本地原生的对话框,但是有些时候,wxWidgets也是一些自己的对话框,比如wxTextEntryDialog,这些对话框统称为一般的对话框.在这一章中,对于那些在各个平台上外观有较大差别的对话框,也给予了单独的图片说明.

我们人为的把标准对话框分成了以下四类:信息对话框,文件和目录对话框,选项和选择对话框以及输入对话框.

8.1信息对话框

在这一小节中,我们来看看以下四种用来提供信息的对话框:wxMessageDialog, wxProgressDialog, wxBusyInfo, and wxShowTip.

wxMessageDialog

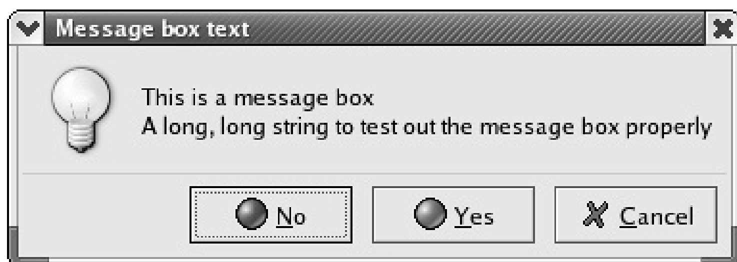
这种对话框显示一个消息和一组按钮,按钮可以为OK, Cancel, Yes或者No,还可以有一个可选的图标,用来显示一个惊叹号或者一个问号.消息文本中还可以包含换行符"\n".

wxMessageDialog::ShowModal函数的返回值用来表征哪个按钮被按下了.

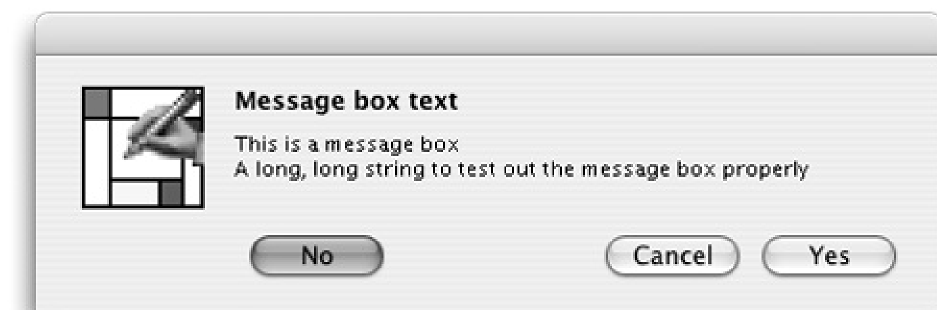
下图显示了wxMessageDialog在windows平台上的样子:



在GTK+平台上:



在Mac平台上:



要创建一个这种对话框,你需要提供父窗口指针,要显示的消息,可选的标题,类型和位置参数,然后调用ShowModal函数显示这个对话框,然后判断这个函数的返回值进行进一步的动作.

其中的类型参数是一个比特位列表,其值如下表所示:

wxOK	显示一个**OK**按钮.
wxCANCEL	显示一个Cancel按钮.
wxYES_NO	显示Yes和No按钮.
wxYES_DEFAULT	设置Yes为默认按钮. 和wxYES_NO一起使用.这是wxYES_NO类型的默认行为.
wxNO_DEFAULT	设置No按钮为默认按钮,和wxYES_NO一起使用.
wxICON_EXCLAMATION	显示一个惊叹号.
wxICON_ERROR	显示一个错误图标.
wxICON_HAND	和wxICON_ERROR相同.
wxICON_QUESTION	显示一个问号.
wxICON_INFORMATION	显示一个信息图标.
wxSTAY_ON_TOP	在windows平台上,这个对话框将在所有的窗口(包括那些其它应用程序的窗口)之上.

wxMessageDialog使用举例

```
#include "wx/msgdlg.h"
wxMessageDialog dialog( NULL, wxT("Message box caption"),
      wxT("Message box text"),
      wxNO_DEFAULT|wxYES_NO|wxCANCEL|wxICON_INFORMATION);
switch ( dialog.ShowModal() )
{
  case wxID_YES:
      wxLogStatus(wxT("You pressed \"Yes\""));
      break;
  case wxID_NO:
      wxLogStatus(wxT("You pressed \"No\""));
      break;
  case wxID_CANCEL:
      wxLogStatus(wxT("You pressed \"Cancel\""));
      break;
  default:
      wxLogError(wxT("Unexpected wxMessageDialog return code!"));
}
```

wxMessageBox

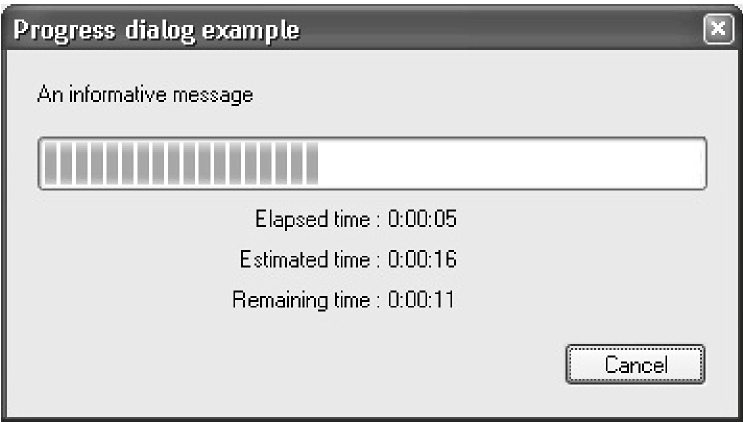
你可以使用更方便的wxMessageBox函数,它的参数为一个消息文本,标题文本,类型和父窗口.例如:

```
if (wxYES == wxMessageBox(wxT("Message box text"),
    wxT("Message box caption"),
    wxNO_DEFAULT|wxYES_NO|wxCANCEL|wxICON_INFORMATION,
    parent))
{
    return true;
}
```

要注意wxMessageBox的返回值和wxMessageDialog::ShowModal的返回值是不一样的,前者返回wxOK, wxCANCEL, wxYES或wxNO,而后者返回wxID_OK, wxID_CANCEL, wxID_YES或 wxID_NO.

wxProgressDialog

wxProgressDialog可以用来显示一个短的消息文本和一个进度条用来指示用户还需要等待多久.它还可以显示一个Cancel 按钮用来中止正在进行的处理,还可以显示已经过去的时间,估计剩余的时间和估计全部的时间.这个对话框是wxWidgets在各个平台上自己实现的.下图显示了这个对话框在windows上的样子:



你既可以用全局变量或者new函数来创建这种对话框,也可以直接使用局部变量创建这种对话框,需要传递的参数包括:标题,消息文本,进度条的最大值,父窗口和类型.

类型的值如下表所示:

wxPD_APP_MODAL	设置为模式对话框.如果没有设置这个类型,对话框为非模式对话框,意味着除了其父窗口以外,应用程序中其它的窗口还可以输入数据.
wxPD_AUTO_HIDE	导致这个进度条对话框在进度达到最大值的时候自动消失.
wxPD_CAN_ABORT	告诉对话框显示一个取消按钮,当用户点击这个按钮以后,下次调用对话框的Update函数将会返回失败.
wxPD_ELAPSED_TIME	显示已逝去时间标签.
wxPD_ESTIMATED_TIME	显示估计全部时间标签.
wxPD_REMAINING_TIME	显示估计剩余时间标签.

在进度对话框被创建以后,其父窗口将被禁用,如果设置了wxPD_APP_MODAL类型,则应用程序中其它的窗口也将被禁用.应用程序应用调用Update函数来更新进度条以及提示信息,如果设置了时间显示标签,则它们的值将被自动计算并在每次调用Update的时候刷新.

如果设置了wxPD_AUTO_HIDE类型,对话框会在进度达到最大值的时候自动隐藏,你应该自己根据其创建方式释放这个对话框.对于由于用户点击取消按钮而导致Update失败的情况,如果你愿意,你可以使用Resume函数还恢复中止的进度条.

wxProgressDialog使用举例

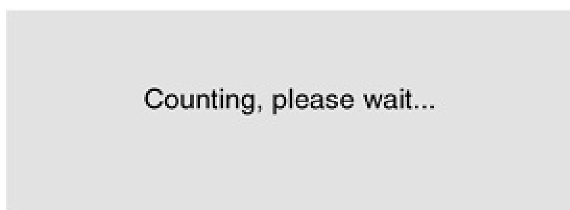
```
#include "wx/progdlg.h"
void MyFrame::ShowProgress()
{
    static const int max = 10;
    wxProgressDialog dialog(wxT("Progress dialog example"),
                            wxT("An informative message"),
                            max,    // range
                            this,   // parent
                            wxPD_CAN_ABORT |
                            wxPD_APP_MODAL |
                            wxPD_ELAPSED_TIME |
                            wxPD_ESTIMATED_TIME |
                            wxPD_REMAINING_TIME);
    bool cont = true;
    for ( int i = 0; i &lt;= max; i++ )
    {
        wxSleep(1);
        if ( i == max )
            cont = dialog.Update(i, wxT("That's all, folks!"));
        else if ( i == max / 2 )
            cont = dialog.Update(i, wxT("Only a half left (very long message)!"));
        else
            cont = dialog.Update(i);

        if ( !cont )
        {
            if ( wxMessageBox(wxT("Do you really want to cancel?"),
                              wxT("Progress dialog question"),
                              wxYES_NO | wxICON_QUESTION) == wxYES )
                break;

            dialog.Resume();
        }
    }
    if ( !cont )
        wxLogStatus(wxT("Progress dialog aborted!"));
    else
        wxLogStatus(wxT("Countdown from %d finished"), max);
}
```

wxBusyInfo

wxBusyInfo其实不是一个对话框不过它的表现和对话框非常相似,当这个对象被创建的时候,屏幕上将显示一个窗口以及一条让用户耐心等待的消息,这个窗口将存在于wxBusyInfo的整个生命周期.在windows平台上,它的长相类似下面的样子:



wxBusyInfo也可以以全局和局部的方式创建,需要传递给构造函数的参数包括一个消息文本和一个父窗口.

wxBusyInfo使用举例

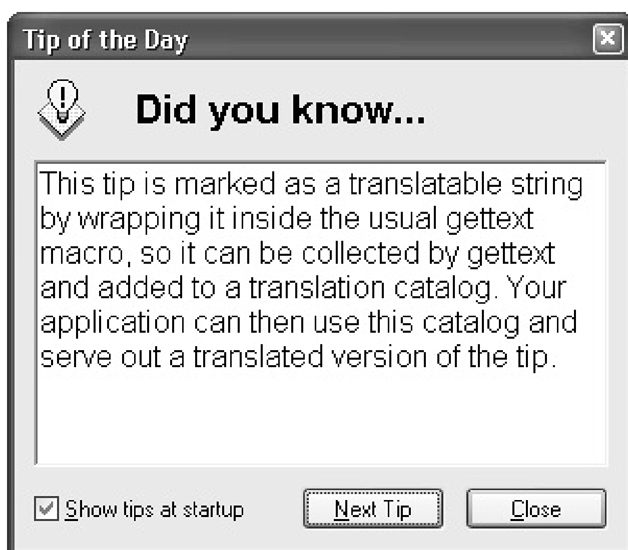
在下面的例子中,首先使用wxWindowDisabler对象禁用应用程序当前创建的所有的窗口,然后显示了一个wxBusyInfo窗口:

```
#include "wx/busyinfo.h"
wxWindowDisabler disableAll;
wxBusyInfo info(wxT("Counting, please wait..."), parent);
for (int i = 0; i &lt; 1000; i++)
{
    DoCalculation();
}
```

wxShowTip

许多应用程序都会在程序启动的时候显示一个附加的窗口,用来给出一些如何使用这个应用程序的提示信息,那些不愿阅读沉闷的文档的人会非常喜欢这样的学习方式的.

启动提示窗口在windows平台上的样子如下图所示:



和大多数对话框不同,这个对话框是使用wxShowTip函数显示的,传递的参数为一个父窗口指针,一个指向wxTipProvider 对象的指针和一个可选的bool参数用来指示是否显示一个复选框,以便用户可以选择是否在应用程序启动的时候显示这个提示框.而函数的返回值则为用户的选择.

你必须实现一个wxTipProvider的派生类,实现其中的GetTip函数才可以使用wxShowTip函数,幸运的是, wxWidgets已经实现了一个这样的基于文本文件的类.你可以直接使用wxCreateFileTipProvider函数,传递以文本文件(每行一个提示文本)路径和默认选择索引来创建一个这样的类.

应用程序应负责在wxTipProvider对象不需要的时候释放这个对象.

wxShowTip使用举例

```
#include "wx/tipdlg.h"
void MyFrame::ShowTip()
{
    static size_t s_index = (size_t)-1;
    if ( s_index == (size_t)-1 )
    {
        // 随机化...
        srand(time(NULL));
        // ...选择一个提示
        s_index = rand() % 5;
    }
    // 传递一个提示文件以及提示索引
    wxTipProvider *tipProvider =
        wxCreateFileTipProvider(wxT("tips.txt"), s_index);
    m_showAtStartup = wxShowTip(this, tipProvider, true);
    delete tipProvider;
}
```

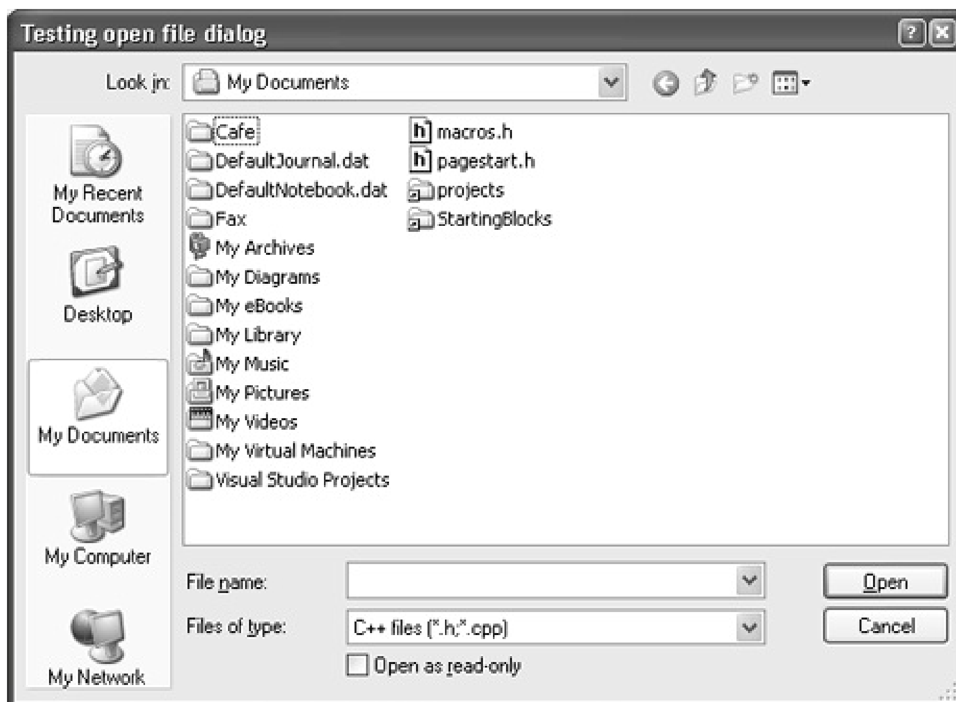
8.2 文件和目录对话框

如果想让用户选择文件和目录,你可以使用下面这两种对话框:wxFileDialog和wxDirDialog.

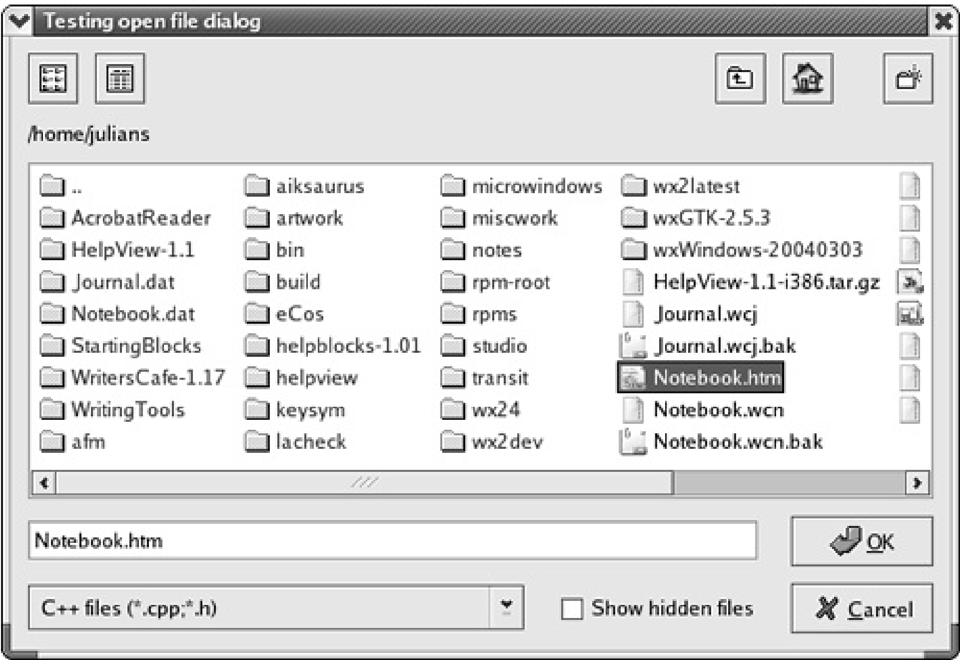
wxFileDialog

wxFileDialog用来让用户选择一个或多个文件.它还有一个专门用来打开文件或者保存文件的变体.

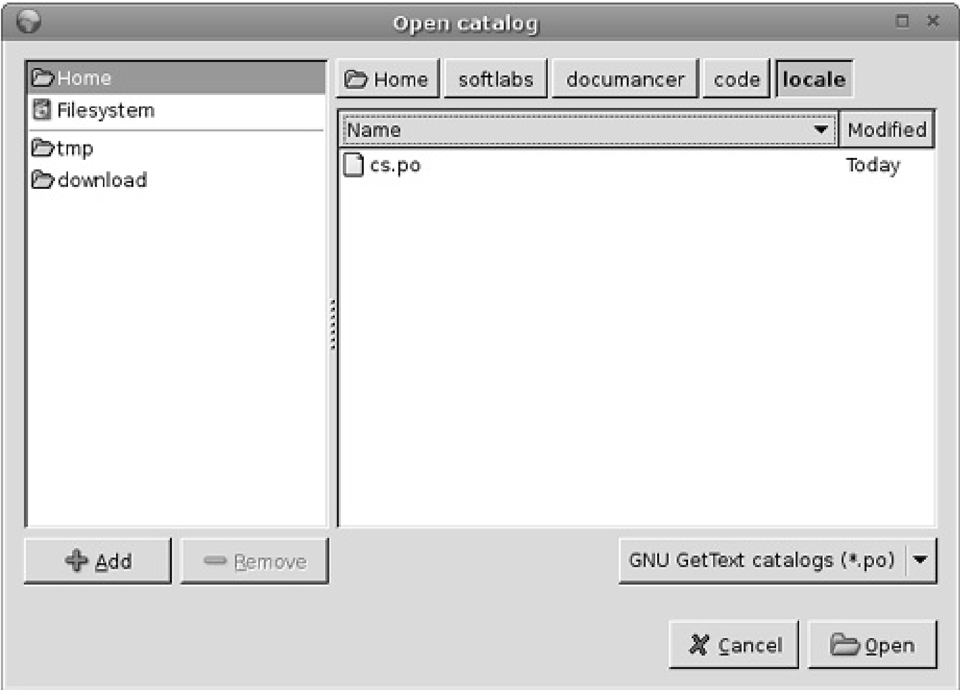
下图演示了windows平台上文件对话框的样子:



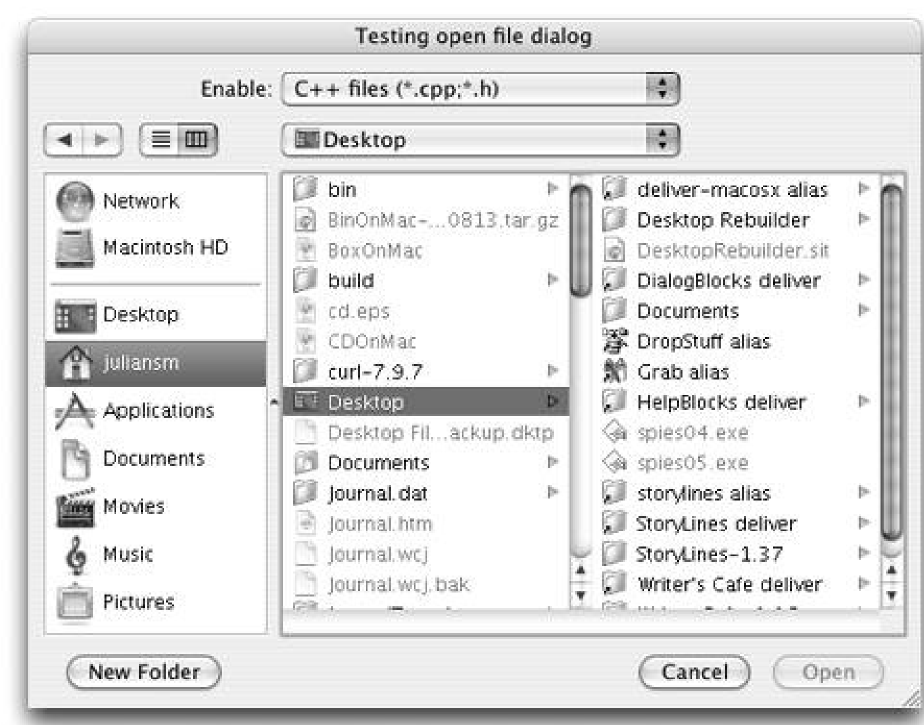
GTK+ 版本:



GTK+ 2.4以上版本:



Mac Os X版本:



创建一个文件对话框需要传递的参数包括一个父窗口,一个显示在对话框标题栏的消息文本,默认的目录,默认文件名,通配符,对话框类型和显示位置(有些平台上忽略这个参数).然后调用其ShowModal函数并且判断函数返回值,如果返回值为wxID_OK则表明用户确认了文件的选择.

目录名和文件名组成一个文件全路径.如果目录名为空,则默认为当前工作目录,如果文件名为空,则没有默认文件名.

通配符用来确定哪种类型的文件应该显示在对话框中.通配符可以用"|"分割多种文件类型,并且可以提供文件类型的描述文字,具体格式如下所示:

```
BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif
```

如果在文件名文本区输入一个带有通配符("*","?")的文件名,然后点确定按钮,将会导致只有符合这个通配名的文件名被显示.

wxFileDialog的类型

如下表所示:

wxSAVE	指定为一个保存文件对话框.
wxOPEN	指定为一个打开文件对话框(默认行为).
wxOVERWRITE_PROMPT	对于保存文件对话框,如果目标文件已经存在,则提示是否覆盖.
wxFILE_MUST_EXIT	用户只能选择已经存在的文件.
wxMULTIPLE	用户可以选择多个文件.

wxFileDialog的成员函数

Getdirectory返回默认的目录名或者单选文件对话框中选中文件的所在的目录名,使用SetDirectory设置默认目录.

GetFilename返回不包括目录部分的默认文件名或者单选文件对话框中选中的文件的文件名. 使用SetFilename设置默认文件名.

GetFilenames使用wxArrayString类型返回多选文件对话框中所有选中的文件名.通常,这些文件名是不含有路径的, 但是在windows平台上,如果选中的是一个快捷方式文件,windows可能会增加上一个全路径,因为应用程序无法通过通过增加当前的目录的方式来得到其全路径.使用GetPaths可以得到包含全路径在内的已选中文件的列表.

GetFilterIndex用来返回默认的基于0的过滤器的索引.过滤器通常以一个下拉框的方式显示.使用SetFilterIndex设置默认的过滤器索引.

GetMessage返回对话框的标题文本. 使用SetMessage函数来设置标题文本.

GetPath以全路径的方式返回单选文件框中默认文件或者选中文件名.对于多选框,应使用GetPaths函数.

GetWildcard返回指定的通配符, SetWildcard用来设置通配符.

wxFileDialog例子

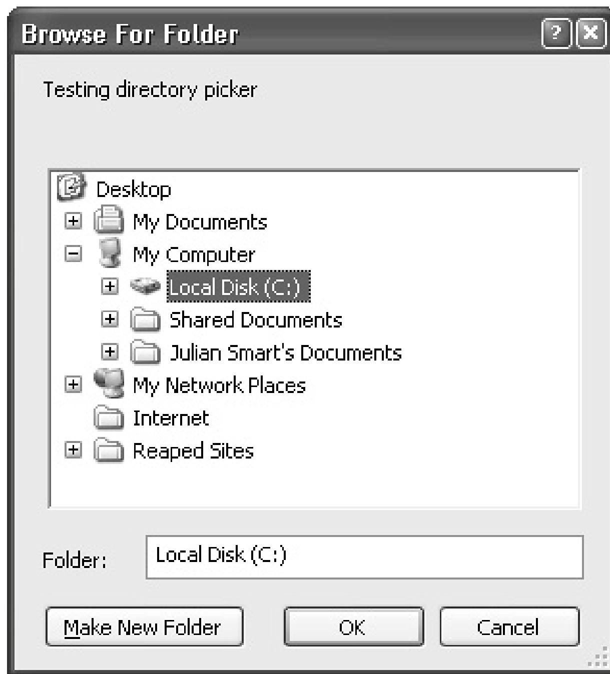
下面的代码用来创建和显示一个用来打开BMP或者GIF类型文件的对话框:

```
#include "wx/filedlg.h"
wxString caption = wxT("Choose a file");
wxString wildcard =
   wxT("BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif");
wxString defaultDir = wxT("c:\\temp"));
wxString defaultFilename = wxEmptyString;
wxFileDialog dialog(parent, caption, defaultDir, defaultFilename,
    wildcard, wxOPEN);
if (dialog.ShowModal() == wxID_OK)
{
    wxString path = dialog.GetPath();
    int filterIndex = dialog.GetFilterIndex();
}
```

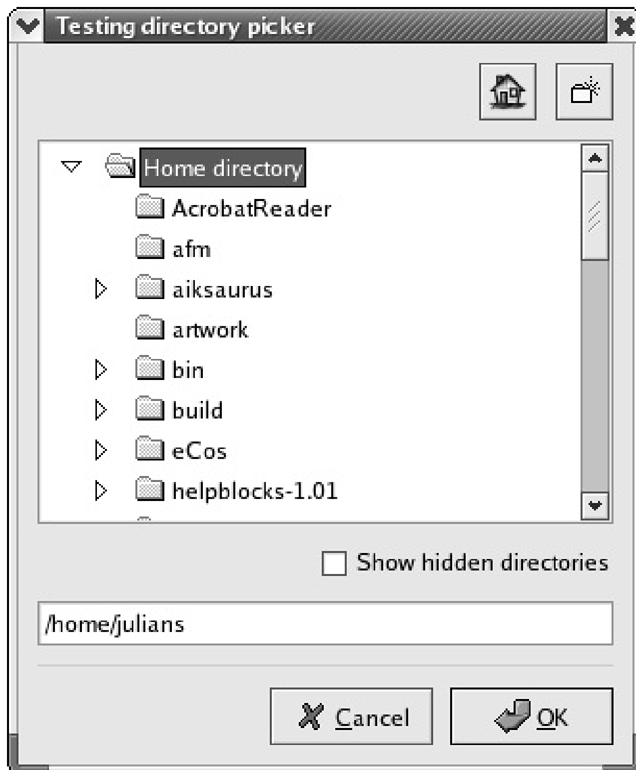
wxDirDialog

wxDirDialog用来让用户选择一个本地或者网络文件夹.如果在构造函数中设置了可选类型wxDD_NEW_DIR_BUTTON,则对话框将显示一个用来创建新文件夹的按钮.

下图演示了windows平台上的目录对话框的样子,这是windows系统提供的原生控件:



linux平台上通常使用GTK+版本的wxDirDialog,如下图所示:



Mac平台上的wxDirDialog和文件选择对话框非常相似,如下图所示:



创建一个目录对话框需要传递的参数包括:一个父窗口,一个标题文本,一个默认路径,一个窗口类型以及一个位置和大小(最后两个参数在某些平台上被忽略),然后调用ShowModal函数,判断返回值是否为wxID_OK以确定用户是否进行了选择.

wxDirDialog成员函数

SetPath和GetPath用来获得和设置用户选择的目录.

SetMessage和GetMessage用来获取和设置标题文本.

wxDirDialog使用举例

```
#include "wx/dirdlg.h"
wxString defaultPath = wxT("/");
wxDirDialog dialog(parent,
    wxT("Testing directory picker"),
    defaultPath, wxDD_NEW_DIR_BUTTON);
if (dialog.ShowModal() == wxID_OK)
{
    wxString path = dialog.GetPath();
    wxMessageBox(path);
}
```

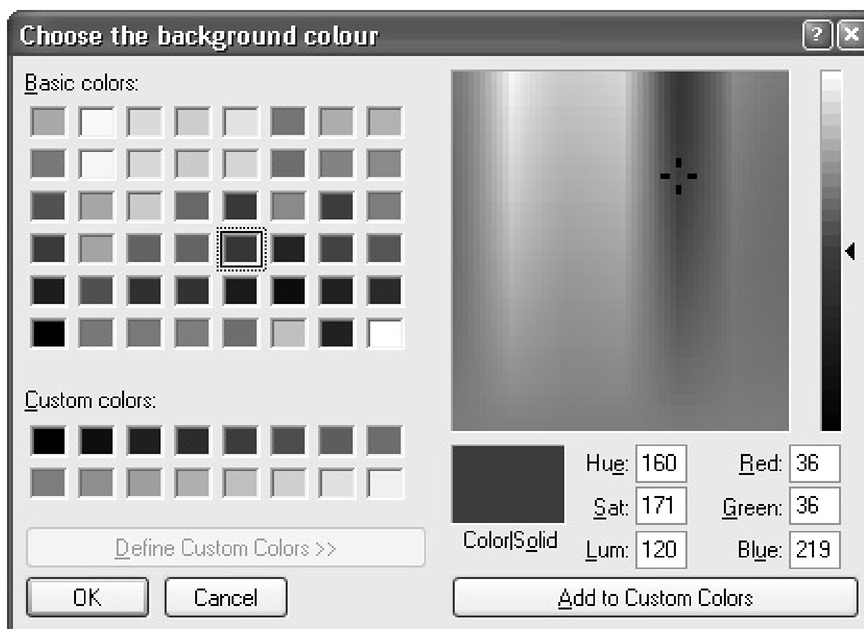
8.3 选择和选项对话框

在这一小节中,我们来看看那些让你作出选择或者允许你给出选项的对话框,包括:wxColourDialog, wxFontDialog, wxSingleChoiceDialog 和 wxMultiChoiceDialog.

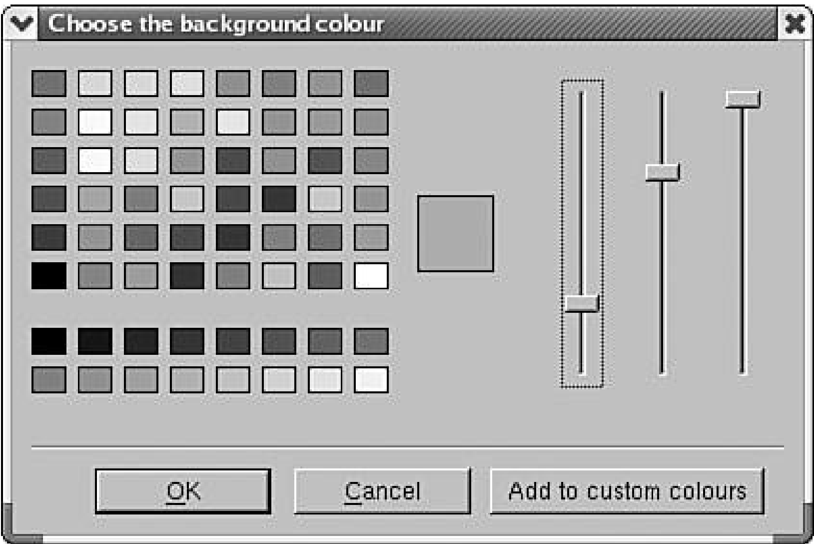
wxColourDialog

这个对话框允许用户从标准颜色或者是一个颜色范围中选择一种颜色.

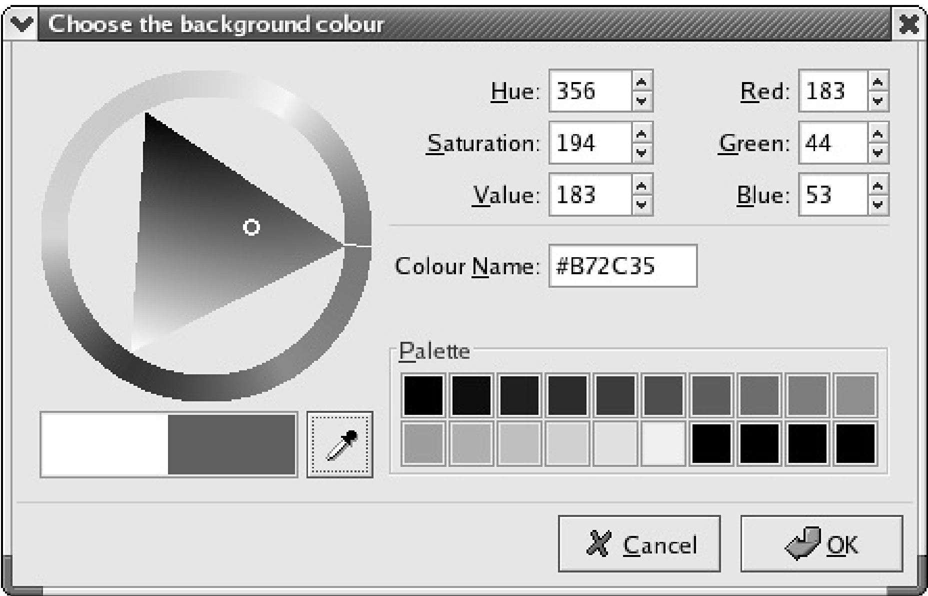
在windows系统上,颜色选择对话框使用的是windows系统提供的原生控件,这个对话框主要包含三个区域,左上角是一个由48中常用颜色组成的区域,左下角是一个拥有16个选项的定制颜色区,用户的应用程序可以自行设置这16种定制的颜色,右边则可以展开用来精确的选择一个颜色.下图演示了这个对话框完全展开以后的样子:



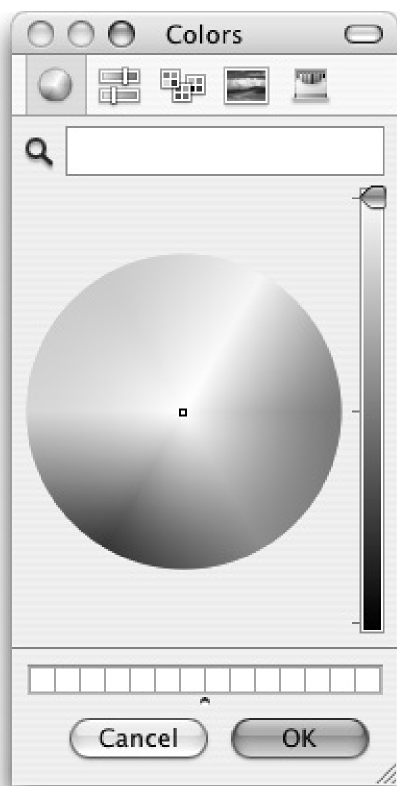
通用的颜色选择对话框如下图所示,可以在GTK+ 1或者X11系统上使用.它包含48个常用颜色和16个可定制颜色,右边包含三个滑条用来选择RGB三原色的值.选好的值用来替换当前选中的定制颜色或者 (如果当前没有选中任何定制颜色的话)第一个定制颜色.和windows系统原生的颜色选择框不同,右边的滑条区域是不可以隐藏的.在Windows平台和其它平台上,你也可以通过wxGenericColourDialog类来使用这个通用颜色选择对话框.



下图演示了GTK+的原生颜色选择对话框:



Mac OsX平台上的颜色选择对话框如下图所示:



要使用颜色选择对话框,你可以创建一个wxColourDialog局部变量,传递的参数包括父窗口指针和一个wxColourData 指针,后者用来设置一些默认数据,然后调用ShowModal函数,当这个函数返回时,通过GetColourData函数获取用户对 wxColourData所做的更改.

wxColourData的成员函数

SetChooseFull定义在windows平台上,颜色选择对话框应该是完全展开的方式显示.否则将只显示左边的部分.GetChooseFull函数则用来获取Bool格式的这个选项的值.

SetColour用来设置默认的颜色, GetColour函数用来返回用户当前选择的颜色.

SetCustomColour使用一个0到15之间的索引和一个wxColour类型的颜色值来设置颜色选择对话框的定制颜色.使用GetCustomColour函数返回当前的可定制范围的颜色值,这个值可能在用户操作颜色对话框的过程中被改变.

wxColourDialog使用举例

下面是一个使用wxColourDialog的例子,它将首先设置wxColourData的各种参数,包括16个灰阶色彩的定制颜色.如果用户没有取消这个对话框,就用用户选择的颜色来设置当前的背景色.

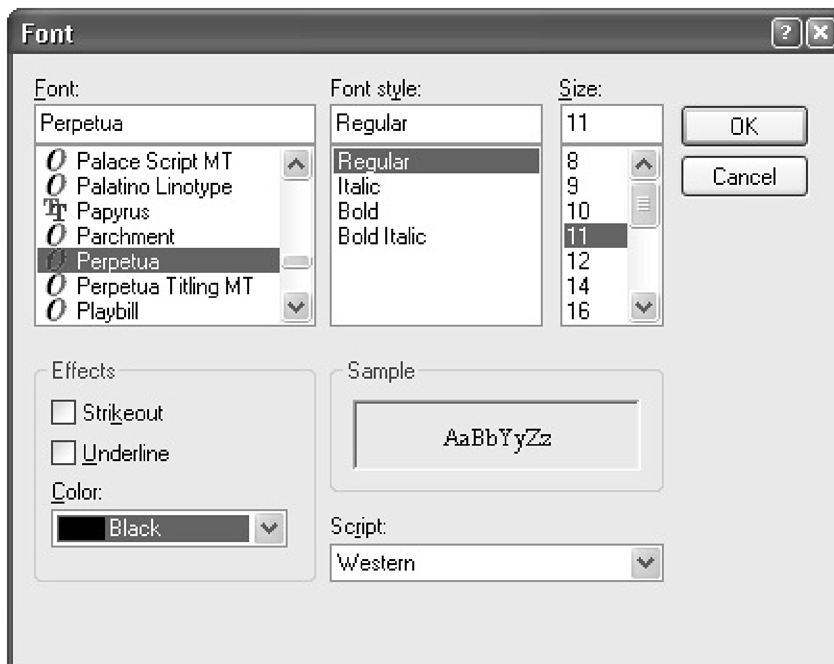
```
#include "wx/colordlg.h"
wxColourData data;
data.SetChooseFull(true);
for (int i = 0; i &lt; 16; i++)
{
    wxColour color(i*16, i*16, i*16);
    data.SetCustomColour(i, color);
}
wxColourDialog dialog(this, &data);
if (dialog.ShowModal() == wxID_OK)
{
    wxColourData retData = dialog.GetColourData();
    wxColour col = retData.GetColour();
    myWindow->SetBackgroundColour(col);
    myWindow->Refresh();
}
```

wxFontDialog

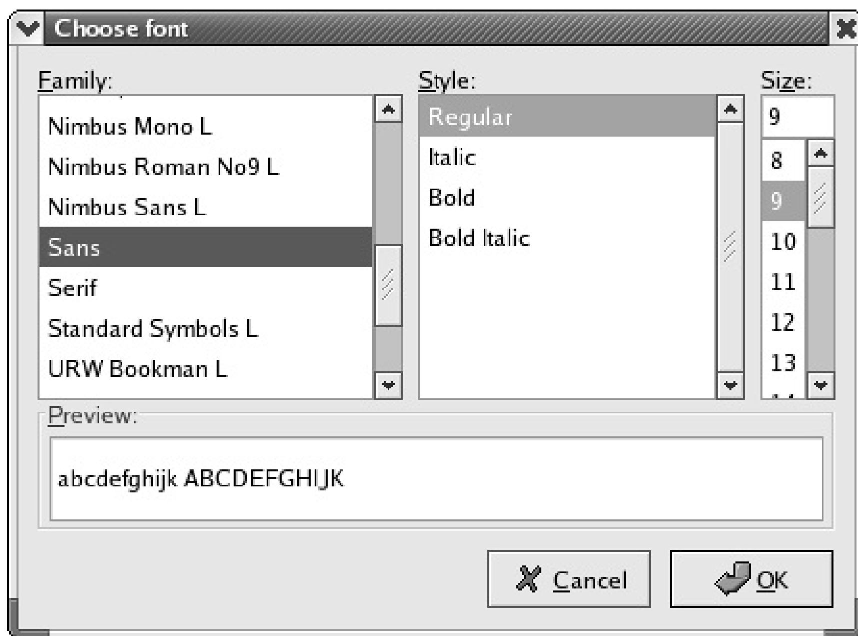
wxFontDialog可以让用户来选择一个字体(在某些平台上,还可以选择字体的颜色).

在Windows平台上,使用的是windows系统提供的原生控件,这个控件给用户的选择包括字体的名称,点大小,类型,粗细,下划线,中划线等属性以及字体的前景颜色.一个白色区域还显示了当前选择字体的样子.注意在windows的字体转换到wxWidgets的字体过程中,中划线属性将被忽略,字体名称则被相应的字体家族名称取代. 而在GTK+平台上,使用的是GTK+的标准字体对话框,这个对话框不允许选择字体前景颜色.

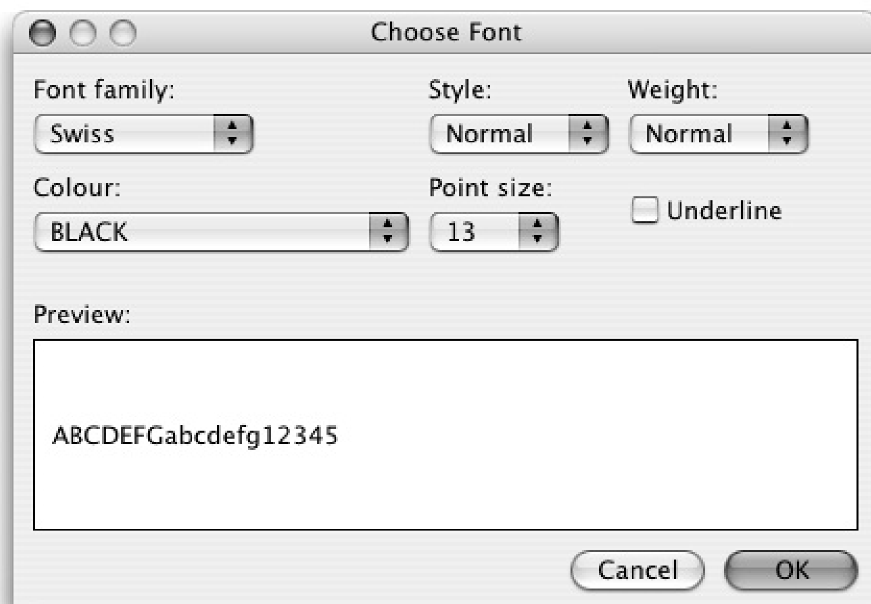
下图演示了windows平台上字体选择对话框的样子:



下图演示了GTK+上的标准字体选择对话框的样子:



除了上述两个平台以外,其它平台字体选择对话框的样子都很相似,允许用户选择的项目包括字体家族名称,大小,类型,粗细,下划线以及前景颜色等,还包括一个示例区用来显示当前字体的样子.这种字体选择框在各种平台都是可以使用的,类的名字为wxGenericFontDialog. 下图演示了Mac平台上的这种字体选择对话框的样子:



要使用字体选择对话框,需要传递的参数包括父窗口和一个wxFontData对象,然后调用其ShowModal函数并且测试其返回值是否为wxID_OK,然后从对话框中获取wxFontData数据,然后视需要调用其GetChosenFont和GetChosenColour函数.

wxFontData的成员函数

EnableEffects函数允许windows平台上的字体选择对话框或者通用版本的字体选择对话框显示颜色和下划线属性控制.在GTK版本上则没有任何效果.GetEnableEffects函数则用来判断当前是否设置了这个标志. 不过即使禁止了这个标志,字体颜色选项还是会被保留.

SetAllowSymbols允许选择符号字体(仅适用于Windows), GetAllowSymbols则返回这个选项的当前设置值.

SetColour设置默认字体颜色, GetColour则返回当前用户选择的颜色.

SetInitialFont返回对话框被创建时候的默认字体. GetChosenFont则返回用户选择的字体(wxFont).

SetShowHelp用来指示应该在字体对话框上显示帮助按钮(仅适用于Windows). GetShowHelp则返回这个选项的当前设置.

SetRange设置用户可以选择的字体大小的范围, 默认值(0, 0)表示任何大小的字体都可以被选择.仅适用于windows.

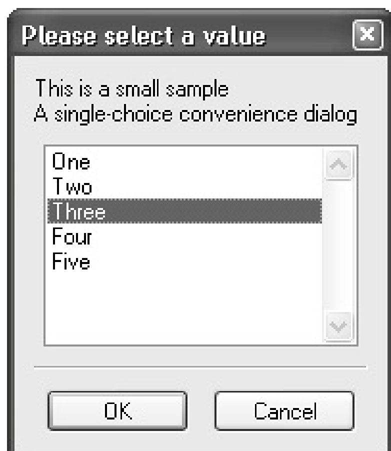
字体选择使用举例

在下面的例子中,应用程序使用字体选择对话框返回的字体在窗口上绘画.

```
#include "wx/fontdlg.h"
wxFontData data;
data.SetInitialFont(m_font);
data.SetColour(m_textColor);
wxFontDialog dialog(this, &data);
if (dialog.ShowModal() == wxID_OK)
{
    wxFontData retData = dialog.GetFontData();
    m_font = retData.GetChosenFont();
    m_textColor = retData.GetColour();
    // 更新当前窗口以便使用当前选择的字体和颜色进行重绘
    myWindow->Refresh();
}
```

wxSingleChoiceDialog

wxSingleChoiceDialog给用户提供了一组字符串以便用户可以选择其中的一个.它的外观如下图所示:



传递给构造函数的参数包括其父窗口指针,一个消息文本,对话框的标题文本以及一个wxArrayString类型的字符串选项列表.其中最后一个参数可以使用一个大小和一个C类型的字符串指针的指针(wxChar**)代替.

SetSelection用来在对话框显示之前设置一个默认的选项,使用GetSelection(返回当前选项的索引)或者GetStringSelection(返回对应的字符串)来在对话框关闭以后获取用户的选择.

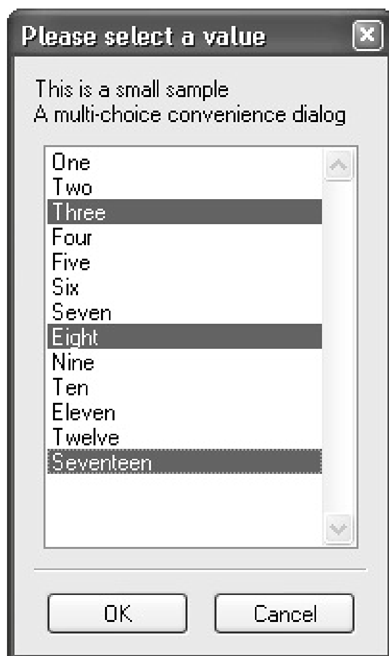
你还可以在这种对话框的构造函数中传递一组char*类型的客户区数据,在对话框被关闭以后,使用GetSelectionClientData函数获得和用户选项对应的客户区数据.

wxSingleChoiceDialog使用举例

```
#include "wx/choicdlg.h"
const wxArrayString choices;
choices.Add(wxT("One"));
choices.Add(wxT("Two"));
choices.Add(wxT("Three"));
choices.Add(wxT("Four"));
choices.Add(wxT("Five"));
wxSingleChoiceDialog dialog(this,
                            wxT("This is a small sample\nA single-choice convenience dial
                            wxT("Please select a value"),
                            choices);
dialog.SetSelection(2);
if (dialog.ShowModal() == wxID_OK)
    wxMessageBox(dialog.GetStringSelection(), wxT("Got string"));
```

wxMultiChoiceDialog

wxMultiChoiceDialog和wxSingleChoiceDialog很相似,不过它允许用户进行多项选择.这种对话框的外观如下图所示:



传递给这种对话框的构造函数的参数包括一个父窗口指针,一个消息文本,对话框标题文本和一个wxArrayString类型的选项列表. 和wxSingleChoiceDialog一样,最后一个参数可以使用字符串指针的列表wxChar**和数量代替.和 wxSingleChoiceDialog不同的是,不可以在构造函数中提供客户区数据.

使用函数SetSelections还设置默认选中的选项,其参数为wxArrayInt类型,代表一个整数数组.使用GetSelections来获取用户的选择,返回值也为wxArrayInt类型.

wxMultiChoiceDialog使用举例

```
#include "wx/choicdlg.h"
const wxArrayString choices;
choices.Add(wxT("One"));
choices.Add(wxT("Two"));
choices.Add(wxT("Three"));
choices.Add(wxT("Four"));
choices.Add(wxT("Five"));
wxMultiChoiceDialog dialog(this,
                            wxT("A multi-choice convenience dialog"),
                            wxT("Please select several values"),
                            choices);
if (dialog.ShowModal() == wxID_OK)
{
    wxArrayInt selections = dialog.GetSelections();
    wxString msg;
    msg.Printf(wxT("You selected %u items:\n"),
        selections.GetCount());

    for ( size_t n = 0; n &lt; selections.GetCount(); n++ )
    {
        msg += wxString::Format(wxT("\t%d: %d (%s)\n"),
                                  n, selections[n],
                                  choices[selections[n]].c_str());
    }
    wxMessageBox(msg, wxT("Got selections"));
}
```

8.4 输入对话框

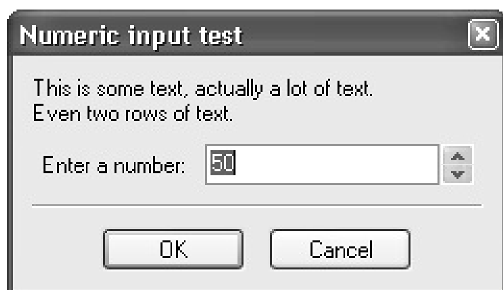
这一类对话框让用户自己输入信息,包括:wxNumberEntryDialog, wxTextEntryDialog, wxPasswordEntryDialog和 wxFindReplaceDialog.

wxNumberEntryDialog

wxNumberEntryDialog提示用户输入一个固定范围内的数字,这个对话框包含一个spin控件因此,用户既可以手动输入数字,也可以通过鼠标点击spin按钮来调整数字的值,这个对话框是wxWidgets自己实现的,因此在各个平台上的表现都是相似的.

创建wxNumberEntryDialog需要提供的参数包括一个父窗口,消息文本,提示文本(显示在spin控件的前面),标题文本,默认值,最小值和最大值,位置等,然后调用ShowDialog函数,如果返回wxID_OK,则可以调用GetValue函数返回用户输入的数字的值.

下图演示了其在windows平台上的样子:



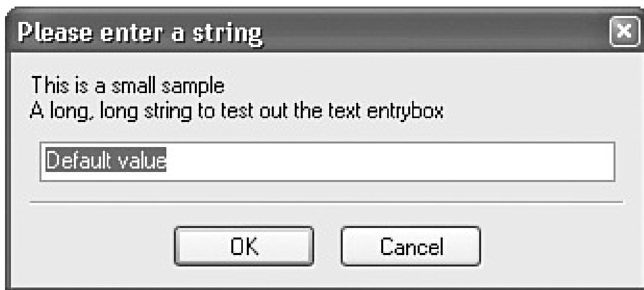
wxNumberEntryDialog使用举例

上图中的对话框是用下面的代码创建的:

```
#include "wx/numdlg.h"
wxNumberEntryDialog dialog(parent,
  wxT("This is some text, actually a lot of text\nEven two rows of text"),
  wxT("Enter a number:"), wxT("Numeric input test"), 50, 0, 100);
if (dialog.ShowModal() == wxID_OK)
{
  long value = dialog.GetValue();
}
```

wxTextEntryDialog和wxPasswordEntryDialog

wxTextEnTRyDialog和wxPasswordEntryDialog提供一个消息文本和一个单行文本框控件,以便用户可以输入文本,它们的功能很类似,只不过在wxPasswordEntryDialog中输入的文本被以掩码的方式显示,因此是不能直接看到的. 下图演示了wxTextEntryDialog对话框在windows平台上的例子:



创建这两个对话框需要提供的参数包括父窗口指针,消息文本,标题文本,默认文本和一个类型参数.类型参数是一个比特位列表,其值为 wxOK, wxCANCEL,wxCENTRE(或者wxCENTER)等,你还可以传递wxTextCtrl的窗口类型wxTE_CENTRE(或 wxTE_CENTER)等.

你可以使用SetValue函数单独设置其默认文本,还可以使用GetValue函数获取用户输入的文本.

wxTextEntryDialog使用举例

上图演示的对话框是用下面的代码创建的:

```
#include "wx/textdlg.h"
wxTextEntryDialog dialog(this,
                           wxT("This is a small sample\n")
                           wxT("A long, long string to test out the text entrybox"),
                           wxT("Please enter a string"),
                           wxT("Default value"),
                           wxOK | wxCANCEL);
if (dialog.ShowModal() == wxID_OK)
    wxMessageBox(dialog.GetValue(), wxT("Got string"));
```

wxFindReplaceDialog

wxFindReplaceDialog是一个非模式对话框,它允许用户用来输入用于搜索的文本以及(如果需要的话)用来替换的文本.实际的搜索动作需要在其派生类或者其父窗口作为这个对话框某个按钮时间的响应来完成.和大多数标准对话框不同,这种对话框必须拥有一个父窗口(非空),并且这个对话框必须是非模式显示的,无论是基于设计还是实现来说.

下图演示了windows系统上的查找和替换对话框:



在其它平台(比如GTK+或Mac OS X)上, wxWidgets使用自己实现的通用版对话框,如下图所示:



处理这个对话框相关的事件

wxFindReplaceDialog对话框在用户点击其上的按钮的时候产生一些命令事件.事件处理函数采用wxFindDialogEvent类型的参数,事件映射宏中的窗口标识符为这个对话框的标识符,这些宏如下表所示:

EVT_FIND(id, func)	**当"查找"按钮被按下时产生.**
EVT_FIND_NEXT(id, func)	当"下一个"按钮被按下时产生.
EVT_FIND_REPLACE(id, func)	当"替换"按钮被按下时产生.
EVT_FIND_REPLACE_ALL(id, func)	当"替换全部"按钮被按下时产生.
EVT_FIND_CLOSE(id, func)	当用户通过取消或者别的途径关闭对话框的时候产生.

wxFindDialogEvent的成员函数

GetFlags返回下列值的一组比特位列表:wxFR_DOWN, wxFR_WHOLEWORD和wxFR_MATCHCASE.

GetFindString返回用户输入的要查找的文本.

GetreplaceString返回用户输入的要替换的文本.

Getdialog返回一个指向产生这个事件的对话框的指针.

向对话框传递数据

创建wxFindReplaceDialog需要传递的参数包括一个父窗口,一个指向wxFindReplaceData的指针,标题文本和一个类型,类型是下表所示比特值的列表:

wxFR_REPLACEDIALOG	**指定对话框是查找替换对话框,而不是查找对话框.**
wxFR_NOUPDOWN	只是查找方向不允许被改变.
wxFR_NOMATCHCASE	支持仅允许大小敏感的搜索或替换.
wxFR_NOWHOLEWORD	指定不支持整字搜索的选项.

wxFindReplaceData保存了所有wxFindReplaceDialog相关的信息.用来对wxFindReplaceDialog对象进行初始化的动作以及用来在wxFindReplaceDialog对话框关闭以后保存其相关的信息,它的值也会在每次产生wxFindDialogEvent事件的时候自动更新,因此你可以直接使用它的成员函数来代替使用wxFindDialogEvent事件的成员函数.使用对话框的GetData函数可以返回在构造对话框的时候填充的wxFindReplaceData对象指针.

wxFindReplaceData的成员函数

下面列出了wxFindReplaceData的用来设置或者获取相关数据的函数,注意那些用于设置的函数只在这个对话框显示之前有用,在对话框显示以后,调用这些用于设置的函数是没有任何效果的.

GetFindString和SetFindString用来设置或者获取要查找的字符串.

GetFlags和SetFlags用来设置或者获取查找替换对话框选项的相应状态(前面已经有具体描述).

GetreplaceString和SetReplaceString用来设置或者获取要替换成的字符串.

查找和替换使用举例

下面演示了查找和替换对话框的使用方法,其中DoFind和DoReplace函数的代码没有列出来,它们用来进行应用程序相关的查找和替换动作.同时,这些函数还应该维护一组应用程序相关的变量,用来保存当前查找的位置,以便下次查找在这之后进行,这些函数还应该完成文档视图相匹配部分的高亮显示.

```
#include "wx/fdrepdlg.h"
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
     EVT_MENU(ID_REPLACE, MyFrame::ShowReplaceDialog)
     EVT_FIND(wxID_ANY, MyFrame::OnFind)
     EVT_FIND_NEXT(wxID_ANY, MyFrame::OnFind)
     EVT_FIND_REPLACE(wxID_ANY, MyFrame::OnReplace)
     EVT_FIND_REPLACE_ALL(wxID_ANY, MyFrame::OnReplaceAll)
     EVT_FIND_CLOSE(wxID_ANY, MyFrame::OnFindClose)
END_EVENT_TABLE()
void MyFrame::ShowReplaceDialog( wxCommandEvent& event )
{
    if ( m_dlgReplace )
    {
        delete m_dlgReplace;
        m_dlgReplace = NULL;
    }
    else
    {
        m_dlgReplace = new wxFindReplaceDialog
                           (
                            this,
                            &m_findData,
                            wxT("Find and replace dialog"),
                            wxFR_REPLACEDIALOG
                           );
        m_dlgReplace->Show(true);
    }
}
void MyFrame::OnFind(wxFindDialogEvent& event)
{
    if (!DoFind(event.GetFindString(), event.GetFlags()))
    {
        wxMessageBox(wxT("No more matches."));
    }
}
void MyFrame::OnReplace(wxFindDialogEvent& event)
{
    if (!DoReplace(event.GetFindString(), event.GetReplaceString(),
        event.GetFlags(), REPLACE_THIS))
    {
        wxMessageBox(wxT("No more matches."));
    }
}
void MyFrame::OnReplaceAll(wxFindDialogEvent& event)
{
    if (DoReplace(event.GetFindString(), event.GetReplaceString(),
        event.GetFlags(), REPLACE_ALL))
    {
        wxMessageBox(wxT("Replacements made."));
    }
    else
    {
        wxMessageBox(wxT("No replacements made."));
    }
}
void MyFrame::OnFindClose(wxFindDialogEvent& event)
{
    m_dlgReplace->Destroy();
    m_dlgReplace = NULL;
}
```

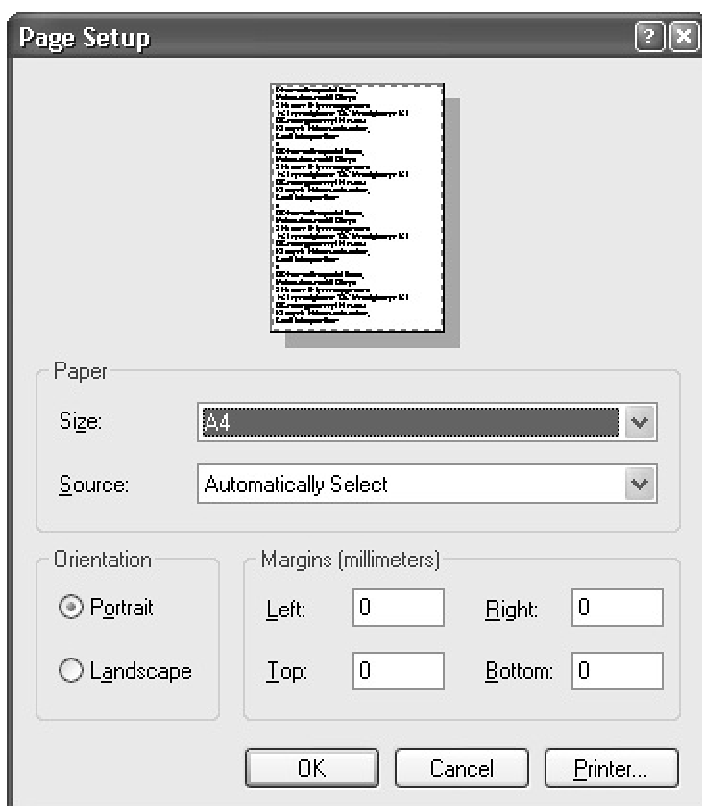
8.5 打印对话框

你可以使用的打印对话框包括wxPageSetupDialog和wxPrintDialog以用来打印文档. 不过,如果你使用wxWidgets的打印框架(包括wxPrintout, wxPrinter以及其它一些类)的话,你的代码中很少需要显式的调用这些对话框.更多关于打印的细节请参考第5章, "绘画和打印."

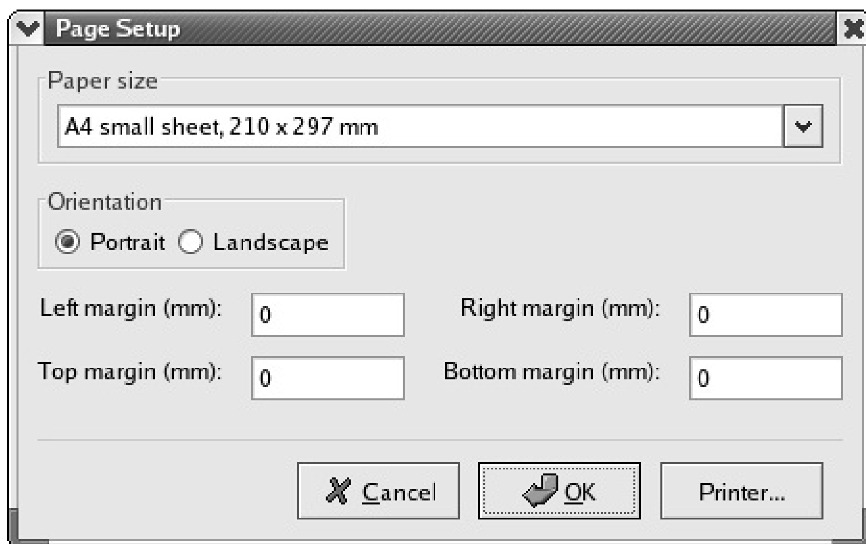
wxPageSetupDialog

wxPageSetupDialog包含一些用来设置纸张大小(A4或者信纸大小等),打印方向(横向或者纵向),以毫米为单位的边框大小等控件还包括一个用来调用另外一个更详细打印设置对话框的按钮.

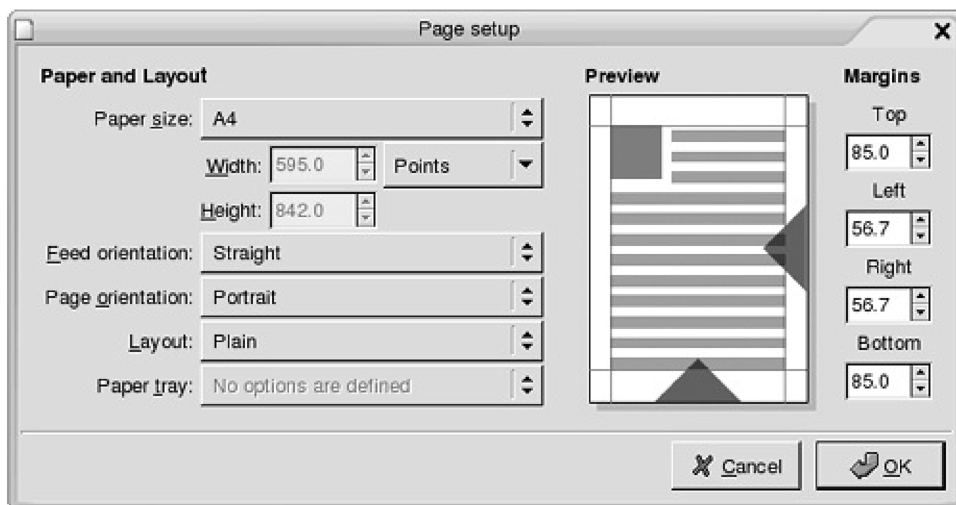
下图演示了wxPageSetupDialog对话框在windows系统上的样子:



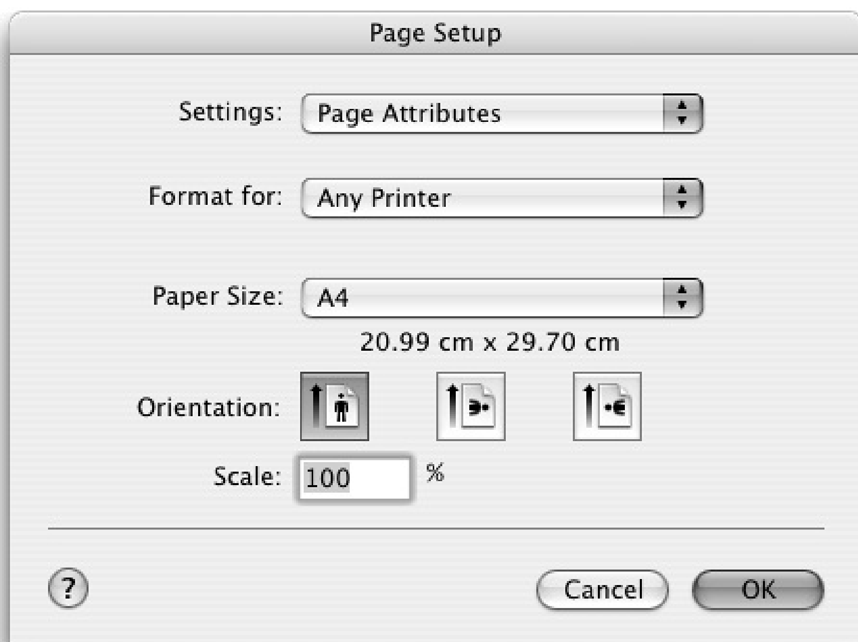
而下面的这个图则是wxWidgets自己实现的使用GTK+的通用的打印设置对话框:



如果使用了Gnome的打印库,则GTK版本中的打印设置对话框将使用Gnome的原生对话框,如下图所示:



Mac版本的打印对话设置对话框图下图所示:



要创建一个打印设置对话框,你需要传递的参数包括一个父窗口和一个指向wxPageSetupDialogData对象的指针,后者用于指定以及从对话框获取打印设置相关的数据,你可以以局部变量或者全局指针的方式创建打印设置对话框.构造函数中的打印设置数据将会被拷贝到打印设置对话框的内部数据中去,而GetPageSetupData函数可以用来获取一个打印设置对话框内部数据的引用.

wxPageSetupData成员函数

Ok函数在其内部数据有效的情况下返回True,在windows平台上,如果没有设置默认的打印机,这个函数可能返回False,在其它平台上,这个函数总是返回True.

SetMarginTopLeft函数使用wxPoint类型的参数以毫米为单位指定打印页面的左边距和上边距. GetMarginTopLeft则用来获取这两个边距.

SetMarginBottomRight函数使用wxPoint类型的参数以毫米为单位设置打印页面的右边距和下边距. GetMarginBottomRight则用来获取这两个边距.

SetPaperId使用标识符来代替具体的大小来设置页面大小. 参考这个函数在手册中的描述来获取相关的标识符. GetPaperId则用来获取当前设置的标识符.

SetPaperSize采用一个wxSize参数来设置以毫米为单位的页面大小. GetPaperSize则用来获取对应的设置.

EnableMargins允许或者禁用对话框上的边界控制控件(仅适用于Windows). GetEnableMargins用来获取这个设置的值.

EnableOrientation用来允许或者禁止对话框上的打印方向控制控件(仅适用于Windows). GetEnableOrientation用来获取这个设置的值.

EnablePaper用来允许或者禁止对话框上选择纸张大小的控件(仅适用于Windows). GetEnablePaper用来获取这个设置的值.

EnablePrinter用来允许或者禁止打印机按钮,这个按钮用来调用另外一个打印设置对话框. GetEnablePrinter用来获取这个设置的值.

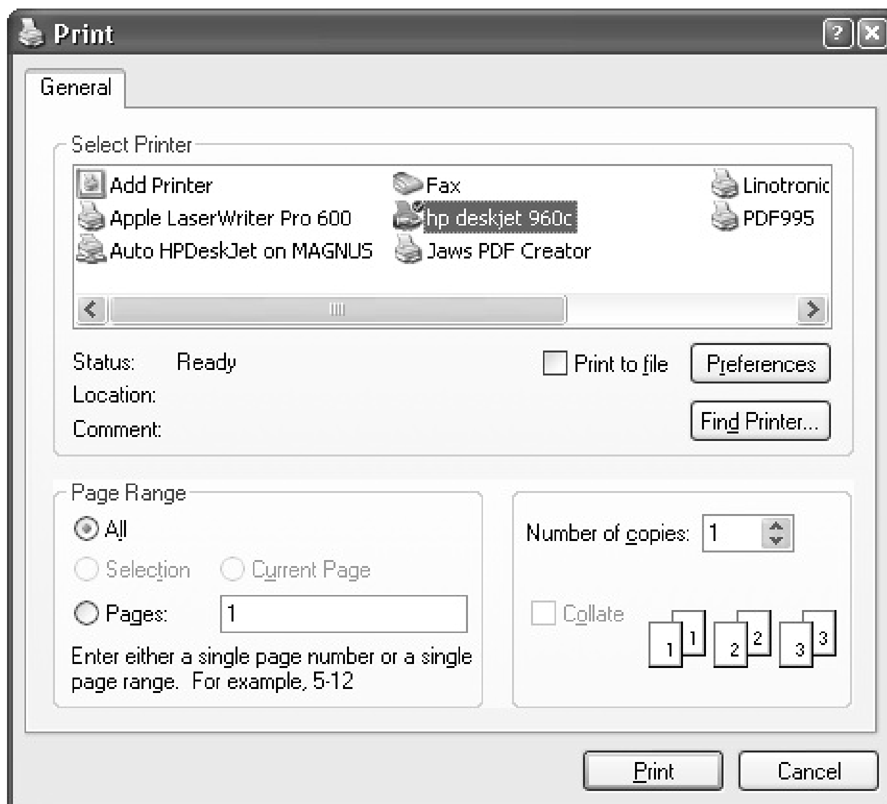
wxPageSetupDialog使用举例

```
#include "wx/printdlg.h"
void MyFrame::OnPageSetup(wxCommandEvent& event)
{
    wxPageSetupDialog pageSetupDialog(this, & m_pageSetupData);
    if (pageSetupDialog.ShowModal() == wxID_OK)
        m_pageSetupData = pageSetupDialog.GetPageSetupData();
}
```

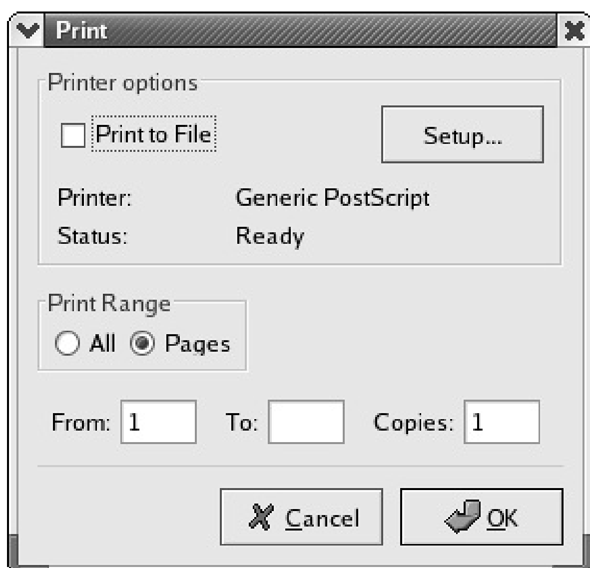
wxPrintDialog

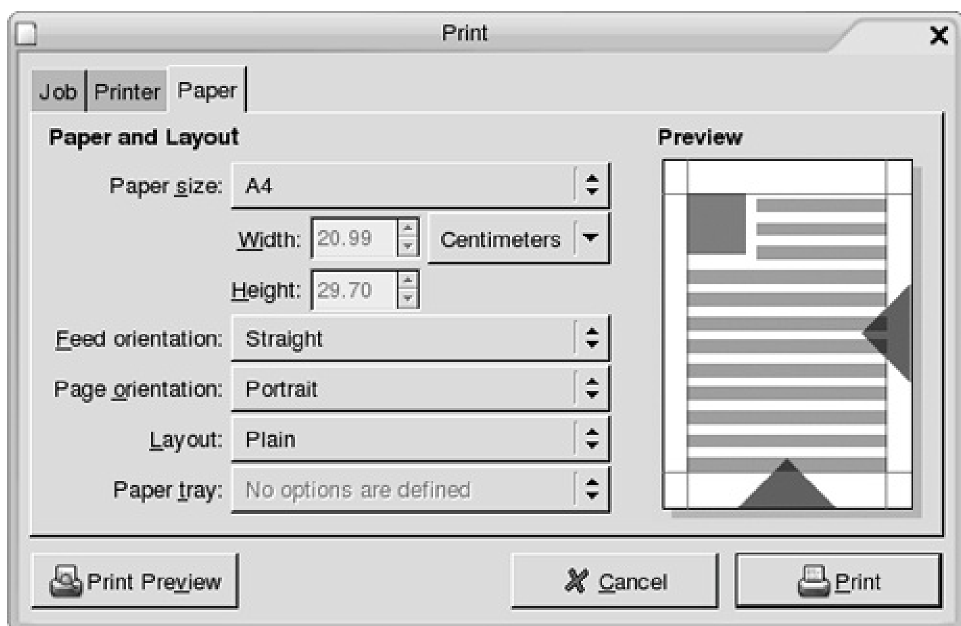
这个对话框用来显式打印及打印设置的标准对话框,当这个对话框关闭的时候你可以从中获得一个wxPrinterDC对象的实例.

下图演示了这个对话框在windows系统上的外观:

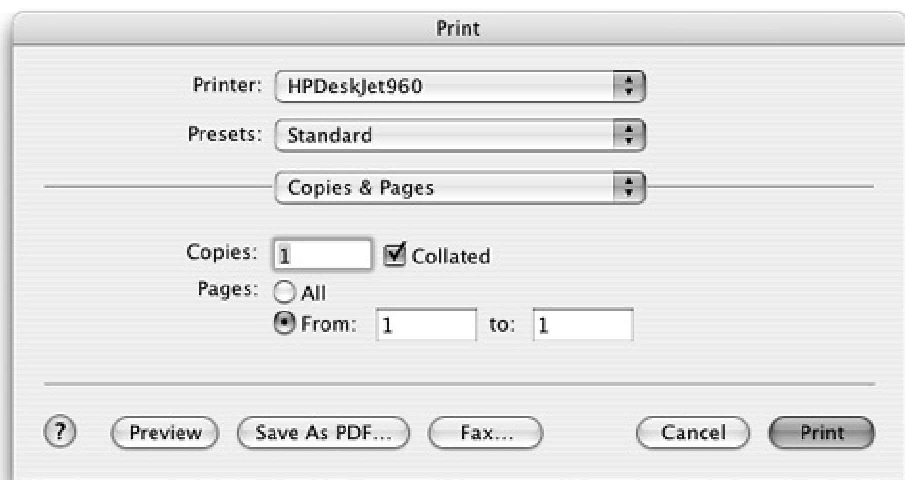


下面的两幅图分别演示了GTK版本中没有使用Gnome打印库和使用了GNome打印库两种情况下对应的这个对话框的样子:





下图则演示了Mac OSX上对应的样子,从图中可以看到,Mac在标准对话框中提供了预览以及存为PDF文件的选项,你可以直接使用这个预览功能.



要创建一个wxPrintDialog对象,你需要提供的参数包括一个父窗口指针,一个wxPrintDialogData对象的指针, 后者的内容将被拷贝给打印对话框内部的对象.如果你希望显示一个打印设置对话框来代替打印对话框,你可以以True为参数调用 wxPrintDialogData:: SetSetupDialog函数,然后再将其传递给wxPrintDialog的构造函数.按照微软的说法,虽然打印设置对话框已经被wxPageSetupDialog所取代,但是一些老的程序可能还在使用以前的标准,因此,这样作可以保证兼容性.

打印对话框被成功关闭的时候,你可以使用GetPrintDialogData函数来获取一个wxPrintDialogData的引用.

调用对话框的GetPrintDC函数来获取一个基于用户选项的打印设备上下文,如果这个函数的返回值不为空,应用程序应该自己释放这个被返回的对象.

Ok函数在打印对话框内部数据有效的时候返回True,在windows平台上,如果没有设置默认的打印机,则Ok返回False,在其它平台上,这个函数总是返回True.

wxPrintDialogData的成员函数

EnableHelp允许或者禁止对话框上的帮助按钮. GetEnableHelp用来获取对应的设置.

EnablePageNumbers允许或者禁止页码设置控件, GetEnablePageNumbers用来获取对应的设置.

EnablePrintToFile允许或者禁止打印到文件按钮. GetEnablePrintToFile用来获取对应的设置.

EnableSelection允许或者禁止用于给用户选择打印范围的单选框. GetEnableSelection用来获取这个设置的值.

SetCollate用来设置Collate复选框的值为True或者false. GetCollate来获取这个复选框的值.

SetFromPage和SetToPage用来设置打印的起始页和终至页. 使用GetFromPage和GetToPage来获取相应的值.

SetMinPage和SetMaxPage用来设置可以打印的最小页数和最大页数. GetMinPage和GetMaxPage则用来获取相应的值.

SetNoCopies用来设置默认打印份数. GetNoCopies用来获取当前设置的打印份数.

SetPrintToFile设置打印到文件的复选框的值. GetPrintToFile则用来获取这个值的当前设定.

SetSelection用来设置打印范围单选框选项. GetSelection则用来返回这个选项的值.

SetSetupDialog用来指示显示打印设置对话框还是打印对话框. GetSetupDialog来获取这个设置.

SetPrintData设置内部的wxPrintData对象. GetPrintData则用来返回内部的wxPrintData对象的一个引用.

wxPrintDialog使用举例

下面的例子演示了怎样显示一个打印对话框以便获取对应的打印上下文:

```
#include "wx/printdlg.h"
void MyFrame::OnPrint(wxCommandEvent& event)
{
    wxPrintDialogData dialogData;
    dialogData.SetFromPage(0);
    dialogData.SetToPage(10);
    wxPrintDialog printDialog(this, & m_dialogData);
    if (printDialog.ShowModal() == wxID_OK)
    {
        // 在调用GetPrintDC()以后, 应用程序
        // 负责管理这个设备上下文
        wxDC* dc = printDialog.GetPrintDC();
        // 在这个设备上下文上绘画
        ...
        // 然后释放它
        delete dc;
    }
}
```

不过,通常你不需要自己直接调用打印对话框 .你应该使用wxWidgets提供的高层打印框架(参考第5章).在你调用wxPrinter::Print函数的时候将会自动显示打印对话框.

第八章小结

在这一章里,你学习了怎样使用标准对话框,以便通过很少的代码来向你的用户提供信息或者从用户那里获得输入.关于更多使用标准对话框的例子,你可以参考wxWidgets自带的samples/dialogs中的例子.在下一章中,我们会介绍一下怎样创建你自己的对话框.

第九章创建定制的对话框

也许很快,也许在将来什么时候,你就需要创建你自己的对话框.也许只是很简单的拥有一些按钮或者文本的对话框,也许是很复杂的包含notebook控件, 多面板,定制控件并且拥有上下文敏感帮助的对话框等.在这一章里,我们将介绍创建定制的对话框以及将数据在对话框控件和C++变量之间传输的一般原理.我们还会介绍一下wxWidgets的资源管理体系,这个体系允许你从XML文件中加载并创建对话框或者是其它的用户界面.

9.1 创建定制对话框的步骤

当你开始创建你自己的对话框的时候,真正有趣的事情才算是刚刚开始.下面是通常你需要采取的步骤:

1. 从wxDialog派生一个新类.
2. 决定数据的存放位置以及应用程序以怎样的方式访问用户选择的数据.
3. 编写代码来创建和布局相关控件.
4. 增加代码以便在C++变量和控件之间进行数据传输.
5. 增加事件映射和相应的处理函数以处理那些来自控件的事件.
6. 增加用户界面更新处理函数,以便将控件设置为正确的状态.
7. 增加帮助,尤其是工具提示以及上下文敏感帮助(在Mac OS中还没有实现),以及在你的应用程序的用户手册中添加你的对话框的使用方法.
8. 在你的应用程序中调用你定制的对话框.

让我们通过一个具体的例子来说明一下这些步骤.

9.2 一个例子:PersonalRecordDialog

正如我们在前面的章节看到的那样,对话框通常有两种:模式对话框和非模式对话框.我们将创建一个定制的模式的对话框,因为这是最常用的一种对话框,而且需要注意的方面也会更少.因为应用程序调用ShowModal函数以后,直到这个函数返回,应用程序将禁止除了这个窗口以及这个窗口产生的别的模式对话框窗口以外的所有别的窗口处理任何用户输入,而将所有的用户交互限制在我们这个模式对话框的小世界里面.

创建一个自定义对话框的许多步骤,可以通过使用一个对话框编辑器而变得简单.比如wxDesigner或者DialogBlocks (译者注:当然,它们都是收费的.一个不收费的开源的对话框编辑器是wxGlade,效果好像也不错).如果使用了对话框编辑器,那么剩余的需要你自己写代码的部分的复杂程度将和你的对话框的复杂程序相关.不过在这里,我们将以全部手写的方式创建这个对话框,以便于演示创建定制对话框的一些基本原理,但是我们强烈推荐你使用任何一种对话框编辑器因为它将节省你大量的时间.

我们将通过这样的一个对话框来演示创建自定义对话框的步骤: 用户需要使用这个对话框输入它的姓名,年龄和性别以及它是否想投票.这个对话框叫做PersonalRecordDialog,它的样子如下图所示:



其中的Reset按钮将所有控件的值复位到它们的默认值.Ok按钮关闭对话框并且以wxID_OK值从ShowModal函数返回. Cancel按钮关闭对话框并且以wxID_CANCEL值从ShowModal函数返回,也不会用用户输入的值来更新对话框的内部变量.而Help按钮则显示一段文本用来大概描述这个对话框(当然,在实际的程序中,这个按钮应该调用一个更漂亮的格式化过的帮助文件).

一个好的用户界面应该不允许用户进行当前上下文中不允许的操作.在这个例子中,如果年龄的值小于18,则投票按钮应该是不可用的(根据英国或者美国的法律).

派生一个新类

下面的代码定义了这个新的对话框类:PersonalRecordDialog,我们使用DECLARE_CLASS宏来提供运行期类型信息,而使用DECLARE_EVENT_TABLE宏来定义了一个事件表.

```
/*!
 * PersonalRecordDialog类定义
 */
class PersonalRecordDialog: public wxDialog
{
    DECLARE_CLASS( PersonalRecordDialog )
    DECLARE_EVENT_TABLE()
public:
    // 构造函数
    PersonalRecordDialog( );
    PersonalRecordDialog( wxWindow* parent,
      wxWindowID id = wxID_ANY,
      const wxString& caption = wxT("Personal Record"),
      const wxPoint& pos = wxDefaultPosition,
      const wxSize& size = wxDefaultSize,
      long style = wxCAPTION|wxRESIZE_BORDER|wxSYSTEM_MENU );
    //用来初始化内部变量
    void Init();
    // 创建窗体
    bool Create( wxWindow* parent,
      wxWindowID id = wxID_ANY,
      const wxString& caption = wxT("Personal Record"),
      const wxPoint& pos = wxDefaultPosition,
      const wxSize& size = wxDefaultSize,
      long style = wxCAPTION|wxRESIZE_BORDER|wxSYSTEM_MENU );
    // 创建普通控件和布局控件
    void CreateControls();
};
```

依照wxWidgets的惯例,我们同时支持了单步创建和两步窗口的方式,单步创建使用的是一个复杂的构造函数,而两步创建则使用的是一个简单的额构造函数和一个复杂的Create函数.

设计数据存储

我们有四个数据要存储:姓名(字符串),年龄(整数),性别(bool)和是否投票(bool).因为我们想用选择框来作为性别的用户界面,所以我们将性别的类型更改为整数类型,但是它可以对外表现为bool类型. 现在让我们来给PersonalRecordDialog类增加这些成员:

```
// 数据成员
wxString    m_name;
int         m_age;
int         m_sex;
bool        m_vote;
// 姓名访问控制
void SetName(const wxString& name) { m_name = name; }
wxString GetName() const { return m_name; }
// 年龄访问控制
void SetAge(int age) { m_age = age; }
int GetAge() const { return m_age; }
// 性别访问控制 (男性 = false, 女性 = true)
void SetSex(bool sex) { sex ? m_sex = 1 : m_sex = 0; }
bool GetSex() const { return m_sex == 1; }
// 是否投票?
void SetVote(bool vote) { m_vote = vote; }
bool GetVote() const { return m_vote; }
```

编码产生控件和布局

现在我们增加一个CreateControls函数来创建控件,这个函数将被Create函数调用,它将增加一个静态文本框,按钮,一个 wxSpinCtrl控件和一个wxTextCtrl控件,一个wxChoice控件,一个wxCheckBox控件,参见上图中的最终效果.

我们使用了布局控件来进行布局,这是为什么它比你想像中的显得复杂了一点的原因(你应该还记得布局控件,我们在第7章对齐进行了描述, 简单的说就是让你的布局拥有随主窗口的大小,语言以及平台的变化而变化的能力).当然你也可以使用别的方法来进行布局,比如你可以使用从wxWidgets资源文件(XRC)中读取布局的方法.

我们来大概刷新一下你关于布局控件的记忆,布局控件把所有的控件分为一个一个的小格子,每个格子刚好可以放置一个控件,这些控件虽然可以拥有非常复杂的布局继承关系,但是它们的窗口继承关系非常单纯,都是继承自同一个父窗口.你可以参考第7章中的第二幅图来刷新你的记忆.

在CreateControls函数中,我们使用了一个垂直盒子布局控件嵌套了另外一个垂直盒子布局控件以便使对话框产生边界.一个水平的盒子布局控件用来放置wxSpinCtrl, wxChoice和wxCheckBox,以及另外一个水平盒子布局控件来放置四个按钮.

```
/*!
 * PersonalRecordDialog控件创建
 */
void PersonalRecordDialog::CreateControls()
{
    // 一个顶层的布局控件
    wxBoxSizer* topSizer = new wxBoxSizer(wxVERTICAL);
    this->SetSizer(topSizer);
    // 第二个顶层布局控件用来产生边界
    wxBoxSizer* boxSizer = new wxBoxSizer(wxVERTICAL);
    topSizer->Add(boxSizer, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
    // 一个友善的提示文本
    wxStaticText* descr = new wxStaticText( this, wxID_STATIC,
        wxT("Please enter your name, age and sex, and specify whether you wish to\nvote i
 a general election."), wxDefaultPosition, wxDefaultSize, 0 );
    boxSizer->Add(descr, 0, wxALIGN_LEFT|wxALL, 5);
    // 空格
    boxSizer->Add(5, 5, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
    // 产生静态文本
    wxStaticText* nameLabel = new wxStaticText ( this, wxID_STATIC,
        wxT("&Name:"), wxDefaultPosition, wxDefaultSize, 0 );
    boxSizer->Add(nameLabel, 0, wxALIGN_LEFT|wxALL, 5);
    // 一个用于输入用户名的文本框
    wxTextCtrl* nameCtrl = new wxTextCtrl ( this, ID_NAME, wxT("Emma"), wxDefaultPosition
 wxDefaultSize, 0 );
    boxSizer->Add(nameCtrl, 0, wxGROW|wxALL, 5);
    // 一个水平布局控件用来放置年龄,性别和是否投票
    wxBoxSizer* ageSexVoteBox = new wxBoxSizer(wxHORIZONTAL);
    boxSizer->Add(ageSexVoteBox, 0, wxGROW|wxALL, 5);
    // 年龄控件的标签
    wxStaticText* ageLabel = new wxStaticText ( this, wxID_STATIC,
        wxT("&Age:"), wxDefaultPosition, wxDefaultSize, 0 );
    ageSexVoteBox->Add(ageLabel, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
    // 用于输入年龄的spin控件
    wxSpinCtrl* ageSpin = new wxSpinCtrl ( this, ID_AGE,
        wxEmptyString, wxDefaultPosition, wxSize(60, -1),
        wxSP_ARROW_KEYS, 0, 120, 25 );
    ageSexVoteBox->Add(ageSpin, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
    // 性别标签
    wxStaticText* sexLabel = new wxStaticText ( this, wxID_STATIC,
        wxT("&Sex:"), wxDefaultPosition, wxDefaultSize, 0 );
    ageSexVoteBox->Add(sexLabel, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
```

```
    // 性别选择框
    wxString sexStrings[] = {
        wxT("Male"),
        wxT("Female")
    };
    wxChoice* sexChoice = new wxChoice ( this, ID_SEX,
        wxDefaultPosition, wxSize(80, -1), WXSIZEOF(sexStrings),
            sexStrings, 0 );
    sexChoice->SetStringSelection(wxT("Female"));
    ageSexVoteBox->Add(sexChoice, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
    // 增加一个可拉升的空白区域
    // 以便让投票选项出现在右边
    ageSexVoteBox->Add(5, 5, 1, wxALIGN_CENTER_VERTICAL|wxALL, 5);
    wxCheckBox* voteCheckBox = new wxCheckBox( this, ID_VOTE,
       wxT("&Vote"), wxDefaultPosition, wxDefaultSize, 0 );
    voteCheckBox ->SetValue(true);
    ageSexVoteBox->Add(voteCheckBox, 0,
        wxALIGN_CENTER_VERTICAL|wxALL, 5);
    // OK和Cancel按钮之间的分割线
    wxStaticLine* line = new wxStaticLine ( this, wxID_STATIC,
        wxDefaultPosition, wxDefaultSize, wxLI_HORIZONTAL );
    boxSizer->Add(line, 0, wxGROW|wxALL, 5);
    // 用来放置四个按钮的水平盒子布局控件
    wxBoxSizer* okCancelBox = new wxBoxSizer(wxHORIZONTAL);
    boxSizer->Add(okCancelBox, 0, wxALIGN_CENTER_HORIZONTAL|wxALL, 5);
    // Reset按钮
    wxButton* reset = new wxButton( this, ID_RESET, wxT("&Reset"),
        wxDefaultPosition, wxDefaultSize, 0 );
    okCancelBox->Add(reset, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
    // Ok按钮
    wxButton* ok = new wxButton ( this, wxID_OK, wxT("&OK"),
        wxDefaultPosition, wxDefaultSize, 0 );
    okCancelBox->Add(ok, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
    // Cancel按钮
    wxButton* cancel = new wxButton ( this, wxID_CANCEL,
        wxT("&Cancel"), wxDefaultPosition, wxDefaultSize, 0 );
    okCancelBox->Add(cancel, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
    // Help按钮
    wxButton* help = new wxButton( this, wxID_HELP, wxT("&Help"),
        wxDefaultPosition, wxDefaultSize, 0 );
    okCancelBox->Add(help, 0, wxALIGN_CENTER_VERTICAL|wxALL, 5);
}
```

数据传输和验证

现在我们已经大概建立起来了这个对话框,但是控件和它的内部成员还没有连接起来,怎样完成这种连接呢?

当对话框第一次显示的时候,wxWidgets会调用InitDialog函数,这个函数产生一个wxEVT_INIT_DIALOG事件.这个事件默认的处理函数是调用transferDataToWindow函数.要把数据从控件传回变量中,你可以在用户确认输入的时候调用 transferDataFromWindow函数,wxWidgets定义了一个默认的wxID_OK命令事件的处理函数,这个函数会在 EndModal函数调用之前调用transferDataFromWindow函数.

因此,你可以通过重载transferDataToWindow和transferDataFromWindow函数以实现数据传输,对于我们这个对话框来说,你可以使用象下面这样的代码:

```
/*!
 * 传输数据到控件
 */
bool PersonalRecordDialog::TransferDataToWindow()
{
    wxTextCtrl* nameCtrl = (wxTextCtrl*) FindWindow(ID_NAME);
    wxSpinCtrl* ageCtrl = (wxSpinCtrl*) FindWindow(ID_SAGE);
    wxChoice* sexCtrl = (wxChoice*) FindWindow(ID_SEX);
    wxCheckBox* voteCtrl = (wxCheckBox*) FindWindow(ID_VOTE);
    nameCtrl->SetValue(m_name);
    ageCtrl->SetValue(m_age);
    sexCtrl->SetSelection(m_sex);
    voteCtrl->SetValue(m_vote);
    return true;
}
/*!
 * 传输数据到变量
 */
bool PersonalRecordDialog::TransferDataFromWindow()
{
    wxTextCtrl* nameCtrl = (wxTextCtrl*) FindWindow(ID_NAME);
    wxSpinCtrl* ageCtrl = (wxSpinCtrl*) FindWindow(ID_SAGE);
    wxChoice* sexCtrl = (wxChoice*) FindWindow(ID_SEX);
    wxCheckBox* voteCtrl = (wxCheckBox*) FindWindow(ID_VOTE);
    m_name = nameCtrl->GetValue();
    m_age = ageCtrl->GetValue();
    m_sex = sexCtrl->GetSelection();
    m_vote = voteCtrl->GetValue();
    return true;
}
```

然而,还有一个更容易的方法.wxWidgets支持验证器,所谓验证器是一个把数据变量和对应的控件联系起来的对象.虽然不总是可以使用,但是只要可以使用,总是可以节省你大量的时间和代码来进行数据的传输和验证.在我们这个例子中,我们可以通过下面的代码来代替上面的额两个函数:

```
FindWindow(ID_NAME)->SetValidator(
      wxTextValidator(wxFILTER_ALPHA, & m_name));
FindWindow(ID_AGE)->SetValidator(
      wxGenericValidator(& m_age));
FindWindow(ID_SEX)->SetValidator(
      wxGenericValidator(& m_sex);
FindWindow(ID_VOTE)->SetValidator(
      wxGenericValidator(& m_vote);
```

这几行代码可以放在CreateControls函数的最后以便代替前面的两个函数,作为一个额外的好处,这样写代码还将增加校验以便用户不可以在姓名框中输入数字.

验证器可以执行两个任务,除了可以传输数据,它们还可以对数据进行校验.并且在数据不合法的时候进行错误提示.在这个例子中,除了姓名框以外,别的输入控件都没有设置校验.wxGenericValidator是一个很简单的验证器,它只负责传输数据,不对数据进行校验,也正因为如此,它支持更多的基本控件.别的验证器则象wxTextValidator一样,提供了各种的校验方法,比如wxTextValidator就可以阻止那些不合法的按键事件传入文本框控件.在这个例子中,我们只是使用了标准校验类型wxFILTER_ALPHA,但是通过验证器的SetIncludes和 SetExcludes函数,我们还可以指定具体哪些字符是允许的哪些是不允许的.

我们还需要略微深入一点的来讨论wxWidgets怎样处理验证过程以便你更好的理解这儿到底发生了什么.正如我们看到的那样,默认的 OnOk函数调用了TransferDataToWindow函数,但是在调用这个函数之前,它还调用了另外一个函数Validate,下面的代码是默认的OnOK的代码:

```
void wxDialog::OnOK(wxCommandEvent& event)
{
    if ( Validate() && TransferDataFromWindow() )
    {
        if ( IsModal() )
            EndModal(wxID_OK); // 如果是模式对话框
        else
        {
            SetReturnCode(wxID_OK);
            this->Show(false); // 如果是非模式对话框
        }
    }
}
```

而默认的Validate函数的实现是遍历对话框所有的直接子窗口(如果设置了wxWS_EX_VALIDATE_RECURSIVELY类型的化,也会调用子窗口的子窗口),依次调用它们的Validate函数,如果任何一个调用失败,则整个Validate过程失败,那么对话框将不会被关闭,并且验证器自己会提供一个消息框以说明失败的原因.

类似的,TransferDataToWindow和TransferDataFromWindow也将自动调用所有子控件的对应的函数.一个验证器可以不做校验,但是必须要可以支持作数据传输.

一个验证器其实是一个事件处理类,wxWidgets的事件处理机制在将事件传递给控件本身之前,会检查这个控件是否被设置了验证器,如果已经设置了,则首先将这个事件传递给验证器,以便验证器可以通过Veto等方法阻止不合法的按键事件.在这种情况下,通常验证器应该调用Beep函数发出一声告警信号以便提示用户某个键已经被按下了但是是非法的.

wxWidgets提供的两种验证器可能是不足够的,尤其是在你想创建自己的控件的时候,这种情况下你可能希望能够创建自己的验证器. 你需要从wxValidator类派生一个新类,这个新类必须带有一个自拷贝的构造函数和一个Clone函数用来返回自己的一份拷贝,还要实现自己的数据传输和验证机制.通常一个验证器都需要存储一个指向某个C++变量的指针,并且构造函数中允许指定不同的模式以实现不同的验证机制.你可以参考 wxWidgets源代码中的include/wx/valtext.h文件和src/common/valtext.cpp文件以了解怎样实现一个自己的验证器,也可以参考第12章,"高级窗口类"中的"制作你自己的控件"这一小节.

处理事件

在这个例子中,wxWidgets默认的处理对于OK和Cancel按钮来说已经是足够,不需要额外添加任何代码,只需要将其设置为对应的wxID_OK和wxID_CANCEL的标识符.不过,在自定义对话框中,进行相应的事件处理是很经常的事情.在这个例子中我们还有一个Reset按钮,当这个按钮被点击时,我们需要复位所有控件的值为它们的默认值.因此我们需要增加OnResetClick事件处理函数,以及一个对应的事件表条目. 这个处理函数的实现是非常简单的,首先我们调用我们自定义的Init成员函数来初始化所有的成员,然后调用TransferDataToWindow函数将数据传输到控件,如下所示:

```
BEGIN_EVENT_TABLE( PersonalRecordDialog, wxDialog )
    ...
    EVT_BUTTON( ID_RESET, PersonalRecordDialog::OnResetClick)
    ...
END_EVENT_TABLE()
void PersonalRecordDialog::OnResetClick( wxCommandEvent& event )
{
    Init();
    TransferDataToWindow();
}
```

处理UI更新

GUI程序员设计代码的一大挑战是确保所有控件在其不可用的时候都是被禁用的.没有什么比弹出一个对话框告诉用户"这个按钮目前是不可用的"更能破坏程序的紧凑性的了.如果一个选项是不可用的,那么它看上去就应该是不可用的,用户点击它或者对它进行任何输入应该都是没有效果的.因此,我们应该在这种情况发生的时候,尽可能快的更新控件的状态使其不可用或者重新可用.

在我们这个例子里面,如果用户输入的年龄小于18岁,我们必须禁用投票按钮,使得这个选项对用户来说是不可用的.你的第一想法可能是要增加一个对于spin控件的事件的更新事件处理函数,并且在其中根据它的值来禁用或者可用投票复选框控件.当然,对于简单的情况来说,这样作完全没有问题但是试想一下,如果有很多种控件存在这种复杂的关联,或者更坏的情况,有时候我们根本不可能在影响某个控件的条件发生改变的时候获得事件通知,比如说我们的剪贴按钮,可能要随着剪贴板的有无数据情况来进行可用或者不可用的操作,而剪贴板是系统变量,它可能受别的应用程序的影响,因为我们自己的应用程序很难在其发生改变的时候收到通知事件.这种情况下唉该怎么办呢?

为了解决这个问题,wxWidgets提供了一个称为wxUpdateUIEvent的事件类,这个事件实在系统空闲的时候(所有其它的事件都已经处理完的时候)发送给应用程序的.你可以使用EVT_UPDATE_UI宏来拦截这个事件,对应于每一个要和别的控件关联改变状态的控件,你需要在事件表中对应增加一个条目,每一个对应的处理函数中,可以检测这个世界当前的状况,从而改变某个控件当前的状态:允许或者禁止,选中或者未选中等, 这种机制允许你将某个控件相关的所有因素在某个特定的时机放在一个函数中处理.你可以松一口气了,因为你不必记住在每一个相关连的事件发生改变的时候去更新对应的控件了.

下面是我们用来处理UI更新事件的相关代码,要注意在代码中我们不能直接判断m_age成员,因为这个成员只有在用户点击了OK按钮以后才会将控件的数据传递过来.

```
BEGIN_EVENT_TABLE( PersonalRecordDialog, wxDialog )
    ...
    EVT_UPDATE_UI( ID_VOTE, PersonalRecordDialog::OnVoteUpdate )
    ...
END_EVENT_TABLE()
void PersonalRecordDialog::OnVoteUpdate( wxUpdateUIEvent& event )
{
    wxSpinCtrl* ageCtrl = (wxSpinCtrl*) FindWindow(ID_AGE);
    if (ageCtrl->GetValue() &lt; 18)
    {
        event.Enable(false);
        event.Check(false);
    }
    else
        event.Enable(true);
}
```

会不会有大量的这种界面更新事件而导致程序的性能降低呢?首先,这种情况是存在的,不过你不必过分担心这个问题,wxWidgets已经作了大量的优化工作,使得这种系统开销将至了最低点.不过如果你的程序真的对性能有很大的要求并且真实感觉是因为这个界面更新的原因导致损失了性能,你还是可以通过wxUpdateUIEvent相关文档中提到的SetMode函数和SetUpdateInterval函数来对这种行为进行进一步的控制, 以减少界面更新事件的消耗的时间.

增加帮助信息

至少有三种帮助信息你可以给你的对话框提供.

- 工具提示
- 上下文敏感帮助
- 联机帮助

当然,你还可以使用更多更先进的没有被wxWidgets显式支持的技术. 我们已经在对话框上边放置了一段用来大概描述这个对话框用途的文本,对于更复杂的对呼框来说,你可以考虑使用wxHtmlWindow代替普通的文本以便可以加载一个HTML文件来提供更复杂的帮助信息.或者,你还可以在每个按钮旁边放置一个小按钮用来提供针对性的帮助信息.

下面我们来说一说wxWidgets明确支持的三种帮助形式:

工具提示

工具提示是在鼠标划过某个控件的时候弹出的一个小窗口,窗口里包含对这个控件用途的一个简短的提示.你可以使用SetToolTip函数来设置控件对应的工具提示.不过,由于对于一些有经验的使用者来说,这个提示可能会显得有些讨厌,因此你应该给你的应用程序增加一个全局的设置以便不显示这些工具提示(这里的意思是说,这个设置使得在对话框创建的时候不调用SetToolTip函数).

上下文敏感帮助

上下文敏感帮助提供的弹出窗口和工具提示窗口是非常相似的,不过这个窗口并不会自动探出来,而需要用户在点击了某个特殊按钮以后再点击对应的控件,或者在某个控件获得焦点的时候按F1键(windows平台适用)的时候才会弹出.在windows平台上,你可以在创建对话框的时候指

定 wxDIALOG_EX_CONTEXTHELP类型以便在标题栏上出现这个上下文敏感帮助按钮,在其它平台上,你可以创建一个 wxContextHelpButton类型的按钮,这个按钮通常应该位于OK和Cancel按钮的旁边.然后,在对话框初始话的时候,增加下面的代码:

```
#include "wx/cshelp.h"
    wxHelpProvider::Set(new wxSimpleHelpProvider);

这将告诉wxWidgets怎样提供上下文敏感帮助,然后调用SetHelpText来设置某个控件的上下文敏感帮助文本.下面是我

void PersonalRecordDialog::SetDialogHelp()
{
    wxString nameHelp = wxT("Enter your full name.");
    wxString ageHelp = wxT("Specify your age.");
    wxString sexHelp = wxT("Specify your gender, male or female.");
    wxString voteHelp = wxT("Check this if you wish to vote.");
    FindWindow(ID_NAME)->SetHelpText(nameHelp);
    FindWindow(ID_NAME)->SetToolTip(nameHelp);
    FindWindow(ID_AGE)->SetHelpText(ageHelp);
    FindWindow(ID_AGE)->SetToolTip(ageHelp);
    FindWindow(ID_SEX)->SetHelpText(sexHelp);
    FindWindow(ID_SEX)->SetToolTip(sexHelp);
    FindWindow(ID_VOTE)->SetHelpText(voteHelp);
    FindWindow(ID_VOTE)->SetToolTip(voteHelp);
}
```

如果你希望自己控制上下文敏感帮助,而不想通过对话框的上下文敏感帮助按钮或者wxContextHelpButton按钮,你可以在任何事件处理函数中使用下面的代码:

```
wxContextHelp contextHelp(window);
```

这将使wxWidgets进入一个检测左键单击的死循环,当单击事件发生时,wxWidgets给对应的控件发送wxEVT_HELP事件,你可以拦截这个事件以便弹出你自己的帮助窗口.

你不必将自己限制于wxWidgets实现的存储显式帮助文本的方式,你可以创建自己的wxHelpProvider派生类,进而实现自己的GetHelp, SetHelp, AddHelp, RemoveHelp和ShowHelp函数.

联机帮助

大多数应用程序都会在一个帮助文件中提供使用应用程序的详细说明.wxWidgets也对应于这种应用提供了几个对应的控件,这些控件都是wxHelpControllerBase的派生类.参考第20章,"优化你的应用程序"来获得这方面更详细的信息.

针对我们这个应用程序而言,我们只是简单的在用户点击帮助按钮的时候显式一个消息框.

```
BEGIN_EVENT_TABLE( PersonalRecordDialog, wxDialog )
    ...
    EVT_BUTTON( wxID_HELP, PersonalRecordDialog::OnHelpClick )
    ...
END_EVENT_TABLE()
void PersonalRecordDialog::OnHelpClick( wxCommandEvent& event )
{
    // 通常我们需要用下面注释的代码提供联机帮助
    /*
    wxGetApp().GetHelpController().DisplaySection(wxT("Personal record dialog"));
     */
    // 在这个例子中,我们只简单的提供一个消息框
    wxString helpText =
      wxT("Please enter your full name, age and gender.\n")
      wxT("Also indicate your willingness to vote in general elections.\n\n")
      wxT("No non-alphabetical characters are allowed in the name field.\n")
      wxT("Try to be honest about your age.");
    wxMessageBox(helpText,
        wxT("Personal Record Dialog Help"),
        wxOK|wxICON_INFORMATION, this);
}
```

完整的例子

这个例子完整的代码列举在附录J,"代码列表"中,你也可以在附送光盘的examples/chap09找到.

调用这个对话框

现在我们需要调用这个对话框来完成所有的编码,我们可以使用下面的代码来调用这个对话框:

```
PersonalRecordDialog dialog(NULL, ID_PERSONAL_RECORD,
    wxT("Personal Record"));
dialog.SetName(wxEmptyString);
dialog.SetAge(30);
dialog.SetSex(0);
dialog.SetVote(true);
if (dialog.ShowModal() == wxID_OK)
{
    wxString name = dialog.GetName();
    int age = dialog.GetAge();
    bool sex = dialog.GetSex();
    bool vote = dialog.GetVote();
}
```

9.3 在小型设备上调整你的对话框

wxWidgets的几个不同的版本都可以用于支持移动设备和嵌入式平台,比如GTK+版本,X11的版本以及WindowsCE的版本(将来也许还会有别的版本)..在这些小型设备使用的最大的问题在于屏幕的大小,一个smartphone的屏幕的大小甚至只有176x220个象素.

对于这样的小型屏幕来说,大多数的对话框都需要一个替代的布局方案.甚至有些控件本身都需要被移除,尤其是那些相比较于桌面系统中其功能已经被移除的那部分控件.你可以使用wxSystemSettings::GetScreenType函数获取当前屏幕的尺寸类型,如下所示:

```
#include "wx/settings.h"
bool isPda = (wxSystemSettings::GetScreenType() &lt;= wxSYS_SCREEN_PDA);
```

这个函数的返回值列举在下表中,因为它随着屏幕实际尺寸的增长而增加,因此你可以直接对其通过简单的整数比较的方法,来判断是否为比某个屏幕更大或者更小的屏幕.

wxSYS_SCREEN_NONE	未定义屏幕类型
wxSYS_SCREEN_TINY	微小屏幕,尺寸小于320x240
wxSYS_SCREEN_PDA	PDA屏幕, 尺寸在320x240到640x480之间
wxSYS_SCREEN_SMALL	小型屏幕,尺寸在640x480到800x600之间
wxSYS_SCREEN_DESKTOP	桌面屏幕,尺寸大于800x600

如果你需要更具体的大小,你可以使用下面的方法:

1. 使用wxSystemSettings::GetMetric, 传递wxSYS_SCREEN_X或wxSYS_SCREEN_Y参数.
2. 使用wxGeTDisplaySize,得到一个wxSize类型的屏幕大小.
3. 创建一个wxDisplay对象,调用其GetGeometry成员函数, 将返回表征当前屏幕的一个矩形区域(wxRect类型).

当你明确知道你的程序将在一个很小的屏幕上运行的时候,你可以作些什么呢?这里我们给出一些建议:

1. 使用从一个不同的XRC文件加载布局的方法来代替整个布局代码或者针对小型的屏幕使用不同的布局代码,只要你不更改控件的类型,事件处理函数的代码就不需要作任何改动.
2. 减小控件以及空白区域的大小.
3. 改变控件的类型(比如使用wxComboBox来代替wxListBox),这将导致一些相关处理函数的代码的改动
4. 改变布局的方向.一些小型设备通常在和桌面屏幕相反的方向拥有更多的空间.

有时候你还需要使用针对特定平台的API函数.比如微软的Smartphone有两个特殊的按钮,你可以对其设置标签,比如"OK", "Cancel"之类的.在这种平台上,你可以使用调用wxDialog::SetLeftMenu和wxDialog::SetRightMenu函数 (参数为标识符,标签以及可选的子菜单)来取代产生两个单独的按钮.当然由于这些函数仅存在于这个平台,因此你需要使用宏开关来区分不同的平台:

```
#ifdef __SMARTPHONE__
    SetLeftMenu(wxID_OK, wxT("OK"));
    SetRightMenu(wxID_OK, wxT("Cancel"));
#else
    wxBoxSizer* buttonSizer = new wxBoxSizer(wxHORIZONTAL);
    GetTopSizer()->Add(buttonSizer, 0, wxALL|wxGROW, 0);
    buttonSizer->Add(new wxButton(this, wxID_OK), 0, wxALL, 5);
    buttonSizer->Add(new wxButton(this, wxID_CANCEL), 0, wxALL, 5);
#endif
```

9.4 一些更深入的话题

下面的这些提示可以让你创建的对话框看上去更专业.

键盘导航

尽量给你的对话框上的控件的标签增加"&"前导符,在某些平台(特别明显的是在windows和GTK+平台上),这将使得用户可以通过键盘在控件之间移动.

总是给你的对话框提供一个取消操作,最好是使用Escape键来执行这个操作.如果你的对话框有一个标识符为wxID_CANCEL的按钮,那么默认情况下它的处理函数将在用户按下Escape键的时候被执行,因此,如果你有一个单纯用来关闭对话框的按钮,最好将它的标识符定义为wxID_CANCEL.

提供一个默认按钮(通常为OK按钮),你可以通过wxButton::SetDefault函数指定一个默认按钮,这个按钮的处理函数将在用户按下回车键的时候被调用.

数据和用户界面分离

为了简化例子,我们在PersonalRecordDialog中采用了把数据存放在对话框内部的方法.然而,一个更好的设计应该是让用户数据和对话框分离,在构造函数中进行赋值操作,这样的话你就可以更方便的传递一组数据给这个对话框的构造函数,以及在用户确认修改的时候(按下OK按钮的时候)从对话框获取一整组数据.这也是大多数标准对话框采用的方法.作为一个练习,你可以使用PersonalRecordData类作为 PersonalRecordDialog的数据成员重写PersonalRecordDialog的代码,以使得对话框的构造函数采用 PersonalRecordData的引用作为参数,而对话框的Getdata函数则返回其内部数据的引用.

一般说来,你应该尽可能的将界面功能和非界面功能分开.这通常会让你的代码显得更紧凑.不要害怕增加新的类,它会让你的设计更优雅,也不要惧怕在类中使用复制或者赋值之类的操作,如果一个对象能够很容易的被赋值和赋值,应用程序可以少些很多底层的赋值代码.

除非你的对话框提供了一个类似"Apply"功能的按钮,否则,在对话框被取消之后,所有内部的数据应该保持原样.使用数据和界面分离的原则也使得实现这一点变得相对容易,因为通常在这种情况下,你只是操作数据的一份拷贝而不是数据本身.

布局

如果你的对话框看上去好像显得有些拥挤或者说有些丑陋,可能是因为你没有给予足够的空白区间.你可以尝试单独使用一个布局控件来增加一个大的边界区域,就象我们在例子中作的那样,在一组控件和另外一组控件之间增加空白,或者使用wxStaticBoxSizer和wxStaticLine 将逻辑上处于两组的控件区隔开.使用wxGridSizer和wxFlexGridSizer布局控件来对齐控件及它们对应的标签以便它们的位置显得不那么零乱.在基于布局控件的布局中,还可以使用空白区域来实现

对齐.比如,通常OK,Cancel按钮和Help按钮应该被设置为右对齐,你可以通过在水平wxBoxSizer的水平方向上增加一个可以缩放的(缩放因子不为负数的)空白区域,然后再增加这些按钮,以便在对话框的水平大小发生改变时实现按钮的右对齐.

尽可能的让你的对话框的边框是可以改变大小的,通常windows系统上的对话框的大小是不能被改变的,但是这样做实在是有些无厘头. 在一个大的对话框上使用很少的控件通常都令用户感到沮丧.在wxWidgets下通过布局控件创建可变大小的对话框是非常简单的事情,你应该尽可能的使用布局控件以便你的对话框可以自适应字体和语言以及对话框大小的改变.当然,你要小心的选择那些可以随着对话框大小的改变而改变大小的控件,例如,多行文本框控件就是一个不错的选择,它的大小随着对话框的大小一起增长可以给用户更多实用的空间,另外,你还可以考虑把增加的空间分给空白区域以实现对齐.注意我们这里说的控件增大,不是只的缩放控件或者是让控件的文本变的更大,而仅仅是指的增加控件的宽度和高度.参考第7章以得到更多关于布局控件的知识.

如果你发现你的对话框上放置了太多的控件,你应该考虑增加面板,并且使用wxNotebook, wxListbook或wxChoicebook来进行分页.使用太多菜单来打开很多互不相干的对话框通常是非常令用户感到不爽的,用分页控件,用户自己选择查看哪个页面,这样就显得方便多了.同时在面板里使用滚动条也是应该被避免的(除非你又充足的原因).这一方面是因为不是所有的平台都支持滚动控件, 另外一方面,这也会给用户一个印象:你对控件布局并没有进行充分的设计.如果你有很多的属性需要用户编辑,你可以考虑使用基于wxGrid的属性编辑器或者其它的第三方控件(参考wxPropertyGrid,在附录E,"wxWidgets的第三方工具"中有提到这个控件).

美学

标签的大小写要保持一致.不要给对话框设置自定义的颜色和字体,这有时候会让你的用户感到抓狂而且也使得你的应用程序看上去有些特立独行 (贬义),和系统当前的风格或者你的应用程序中的其它对话框不一致.要在各个平台都取得不错的效果,最好让wxWidgets自己决定对话框的颜色或者字体.取而代之的,你可以把精力花费在设计一些新颖而紧凑的小图片上,并且在合适的位置使用wxStaticBitmap控件显式它.

对话框的替代品

最后,对于那些非模式的解决方案,你应该好好考虑是否要使用一个对话框,也许在应用程序的主窗口中使用TAB控件会是更好的选择. 尽管我们前面所说的大部分原则都适用于非模式的对框框,但是它们确实有一些不同之处,比如你需要额外考虑布局(通常这种窗口拥有原少于它的大小的控件)以及数据同步(对于非模式窗口来说,它在显示的时候不能以独占的方式访问它的数据了).

9.5 使用wxWidgets资源文件

你可以从一个Xml文件中加载对话框,frame窗口,菜单条,工具条等等,而不一定非要用C++代码来创建它们.这更符合界面和代码分离的原则,它可以让应用程序的界面在运行期改变.XRC文件可以通过一系列用户界面设计的工具导出,比如:wxDesigner, DialogBlocks, XRCed和wxGlade.

加载资源文件

要使用XRC文件,你需要在你的代码中包含wx/xrc/xmlres.h头文件.

如果你打算将你的XRC文件转换成二进制的XRS文件(我们很快会介绍到),你还需要增加zip文件系统的处理函数,你可以在你的OnInit函数中增加下面的代码来作到这一点:

```
#include "wx/filesys.h"
#include "wx/fs_zip.h"
wxFileSystem::AddHandler(new wxZipFSHandler);
```

首先初始化XRC处理系统,你需要在OnInit中增加下面的代码:

```
wxXmlResource::Get()->InitAllHandlers();
```

然后加载一个XRC文件:

```
wxXmlResource::Get()->Load(wxT("resources.xrc"));
```

这只是告诉wxWidgets这个资源文件的存在,要创建真实的用户界面,还需要类似下面的代码:

```
MyDialog dlg;
wxXmlResource::Get()->LoadDialog(& dlg, parent, wxT("dialog1"));
dlg.ShowModal();
```

下面的代码则演示了怎样创建菜单条,菜单,工具条,位图,图标以及面板:

```
MyFrame::MyFrame(const wxString& title): wxFrame(NULL, -1, title)
{
    SetMenuBar(wxXmlResource::Get()->LoadMenuBar(wxT("mainmenu")));
    SetToolBar(wxXmlResource::Get()->LoadToolBar(this,
                                              wxT("toolbar")));
    wxMenu* menu = wxXmlResource::Get()->LoadMenu(wxT("popupmenu"));
    wxIcon icon = wxXmlResource::Get()->LoadIcon(wxT("appicon"));
    SetIcon(icon);
    wxBitmap bitmap = wxXmlResource::Get()->LoadBitmap(wxT("bmp1"));
    // 既可以先创建实例再加载
    MyPanel* panelA = new MyPanel;
    panelA = wxXmlResource::Get()->LoadPanel(panelA, this,
                                                   wxT("panelA"));
    // 又可以直接创建并加载
    wxPanel* panelB = wxXmlResource::Get()->LoadPanel(this,
                                                   wxT("panelB"));
}
```

wxWidgets维护一个全局的wxXmlResource对象,你可以直接拿来使用,也可以创建一个你自己的 wxXmlResource对象,然后加载某个资源文件,然后使用和释放它.你还可以使用wxXmlResource::Set函数来让应用程序用某个 wxXmlResource对象来取代全局资源对象,并释放掉那个旧的.

要为定义在资源文件中的控件定义事件表条目,你不能直接使用整数的标识符,因为资源文件中存放的其实是字符串,你需要使用XRCID 宏,它的参数是一个资源名,返回值是这个资源对应的标识符.其实XRCID就是直接使用的wxXmlResource::GetXRCID函数,举例如下:

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(XRCID("menu_quit"),  MyFrame::OnQuit)
    EVT_MENU(XRCID("menu_about"), MyFrame::OnAbout)
END_EVENT_TABLE()
```

使用二进制和嵌入式资源文件

你可以把多个wxWidgets资源文件编译成一个二进制的压缩的xrs文件.使用的工具wxrc可以在wxWidgets的utils/wxrc目录中找到,使用方法如下:

```
wxrc resource1.xrc resource2.xrc -o resource.xrs
```

使用wxXmlResource::Load函数加载一个二进制的压缩的资源文件,和加载普通的文本Xml文件没有区别.

提示: 你可以不必把你的XRC文件单独制作一个zip压缩文件,而是把它放在其它一个可能包含HTML文件以及图片文件的普通的zip压缩文件中, wxXmlResource::Load函数支持虚拟文件系统定义(参考第14章:"文件和流"),因此你可以通过下面的方法来加载压缩文件中的XRC文件:

```
wxXmlResource::Get()->Load(wxT("resources.bin#zip:dialogs.xrc"));
```

你也可以将XRC文件编译为C++的代码,通过和别的C++的代码编译在一起,你就可以去掉某个单独的资源文件了.编译用的命令行如下所示:

```
wxrc resource1.xrc resource2.xrc c -o resource.cpp
```

编译方法和编译普通的C++代码相同,这个文件包含一个InitXmlResource函数,你必须在你的主程序中调用这个函数:

```
extern void InitXmlResource(); // defined in generated file
wxXmlResource::Get()->InitAllHandlers();
InitXmlResource();
```

下面列出了wxrc程序的命令行参数:

短命令格式	长命令格式	描述
-h	help	显式帮助信息.
-v	verbose	打印执行过程信息.
-c	cpp-code	编译目标为C++代码,而不是XRS文件.
-p	python-code	编译目标为Python代码而不是XRS文件.
-e	extra-cpp-code	和-c一起使用,指示为XRC定义的窗口生成头文件.
-u	uncompressed	不要压缩Xml文件(C++ only).
-g	gettext	将相关的字符串翻译为poEdit或者gettext可以识别的格式.输出到标准输出或者某个文件中(如果指定了-o参数的话).
-n	function <name>	指定特定的C++初始化函数(和-c一起使用).
-o <filename>	output <filename>	指定输出文件名,比如resource.xrs or resource.cpp.
-l <filename>	list-of-handlers <filename>	列举这个资源文件所需要的处理函数.

资源翻译

如果wxXmlResource对象创建的时候指定了wxXRC_USE_LOCALE标记(默认行为),所有可显示的字符串都将被认为是需要翻译的,具体内容参考第16章,"编写国际化应用程序",然后poEdit并能查找XRC文件来发现那些需要翻译的字符串,因此,必须使用"-g" 参数产生一个对应的C++文件,以供poEdit使用,命令行如下:

```
wxrc -g resources.xrc -o resource_strings.cpp
```

然后你就可以使用poEdit来搜索这个和其它的C++文件了.

XRC的文件格式

这里显然不是完整描述XRC文件格式的地方,因此我们只举一个简单的使用了布局控件的例子:

```
<?xml version="1.0"?>
<resource version="2.3.0.1">
<object class="wxDialog" name="simpledlg">
    <title>A simple dialog</title>
    <object class="wxBoxSizer">
      <orient>wxVERTICAL</orient>
      <object class="sizeritem">
        <object class="wxTextCtrl">
          <size>200,200d</size>
          <style>wxTE_MULTILINE|wxSUNKEN_BORDER</style>
          <value>Hello, this is an ordinary multiline\n textctrl....</value>
        </object>
        <option>1</option>
        <flag>wxEXPAND|wxALL</flag>
        <border>10</border>
      </object>
      <object class="sizeritem">
        <object class="wxBoxSizer">
          <object class="sizeritem">
            <object class="wxButton" name="wxID_OK">
              <label>Ok</label>
              <default>1</default>
            </object>
          </object>
          <object class="sizeritem">
            <object class="wxButton" name="wxID_CANCEL">
              <label>Cancel</label>
            </object>
            <border>10</border>
            <flag>wxLEFT</flag>
          </object>
        </object>
        <flag>wxLEFT|wxRIGHT|wxBOTTOM|wxALIGN_RIGHT</flag>
        <border>10</border>
      </object>
    </object>
  </object>
</resource>
```

XRC文件格式的详细描述可以在wxWidgets自带的文档目录docs/tech/tn0014.txt中找到.如果你使用对话框编辑器的话,你管它的文件格式干嘛呢.

你可能回问怎样在XRC文件中指定二进制的图片或者图标文件呢?实际上这些资源是通过URLs来指定的,wxWidgets的虚拟文件系统将会从合适的地方(比如一个压缩文件中)获取指定的文件.举例如下:

```
<object class="wxBitmapButton" name="wxID_OK">
  <bitmap>resources.bin#zip:okimage.png</bitmap>
</object>
```

关于使用虚拟文件系统加载资源或者图片的细节,请参考第10章,"在程序中使用图片"以及第14章"文件和流".

编写资源处理类

XRC系统使用不同的资源处理类来识别Xml文件中定义的不同的资源.如果你编写了自己的控件,你就需要编写自己的资源处理类.

wxButton的资源处理类如下所示:

```
#include "wx/xrc/xmlres.h"
class wxButtonXmlHandler : public wxXmlResourceHandler
{
DECLARE_DYNAMIC_CLASS(wxButtonXmlHandler)
public:
    wxButtonXmlHandler();
    virtual wxObject *DoCreateResource();
    virtual bool CanHandle(wxXmlNode *node);
};
```

资源处理类的实现是非常简单的.在其构造函数的实现中,使用 XRC_ADD_STYL宏来使得处理类可以识别控件相关的特殊的窗口类型,然后使用AddWindowStyles增加这些类型.然后在DoCreateResource函数中,使用两步法创建按钮实例,其中第一步要使用XRC_MAKE_INSTANCE函数,然后调用Create函数,参数需要使用对应的函数从Xml文件中获得.而CanHandle函数则用来回答是否这个处理类可以处理某个Xml节点的问题.使用一个处理类处理多种Xml节点是允许的.

```
IMPLEMENT_DYNAMIC_CLASS(wxButtonXmlHandler, wxXmlResourceHandler)
wxButtonXmlHandler::wxButtonXmlHandler()
: wxXmlResourceHandler()
{
    XRC_ADD_STYLE(wxBU_LEFT);
    XRC_ADD_STYLE(wxBU_RIGHT);
    XRC_ADD_STYLE(wxBU_TOP);
    XRC_ADD_STYLE(wxBU_BOTTOM);
    XRC_ADD_STYLE(wxBU_EXACTFIT);
    AddWindowStyles();
}
wxObject *wxButtonXmlHandler::DoCreateResource()
{
   XRC_MAKE_INSTANCE(button, wxButton)
   button->Create(m_parentAsWindow,
                    GetID(),
                    GetText(wxT("label")),
                    GetPosition(), GetSize(),
                    GetStyle(),
                    wxDefaultValidator,
                    GetName());
    if (GetBool(wxT("default"), 0))
        button->SetDefault();
    SetupWindow(button);
    return button;
}
bool wxButtonXmlHandler::CanHandle(wxXmlNode *node)
{
    return IsOfClass(node, wxT("wxButton"));
}
```

要使用某种处理类,应用程序需要包含相应的头文件并且登记这个处理类,就象下面这样:

```
#include "wx/xrc/xh_bttn.h"
wxXmlResource::AddHandler(new wxBitmapXmlHandler);
```

外来控件

XRC文件还可以通过class="unknown"来指定某个控件是外来的或者说是"未知的"控件.这可以用来实现在其父窗口已经加载到应用程序之中以后,使用C++代码来创建这个未知的控件.当XRC文件加载一个未知控件的时候,它将创建一个用来占位的窗口,然后在代码中,可以使用C++先创建这个实际的控件,然后使用AttachUnknownControl函数替换掉那个用来占位的窗口.如下所示:

```
wxDialog dlg;
// 加载对话框
wxXmlResource::Get()->LoadDialog(&dlg, this, wxT("mydialog"));
// 创建特殊控件
MyCtrl* myCtrl = new MyCtrl(&dlg, wxID_ANY);
// 增加到对话框里
wxXmlResource::Get()->AttachUnknownControl(wxT("custctrl"), myCtrl);
// 显示整个对话框
dlg.ShowModal();
```

外来的控件在XRC文件中可以这样定义:

```
<object class="unknown" name="custctrl">
  <size>100,100</size>
</object>
```

使用这种技术,你可以既不用创建新的资源处理类,又可以在资源文件中使用未知的控件.

第九章小结

在这一章里,你学习了怎样创建自定义对话框的一些基本原理,包括布局控件概览,使用验证器以及处理UI更新时间的优点等.在wxWidgets自带的 samples/dialogs目录中你也可以看到一个自定义对话框的例子.另外samples/validate目录中的例子则演示了一般的文本验证器的用法.在下一章里,我们来看看怎样处理图片.

第十章使用图像编程

这一章来了解一下我们可以使用图片来作些什么事情.一幅图胜过千言万语,在wxWidgets,工具条,树形控件,notebooks,按钮,Html窗口和特定的绘画代码中,都会用到图片.有时候它们还会在不可见的地方发挥作用,比如我们可以用它来创建双缓冲区以避免闪烁.这一章里,我们会接触到各种各样的图片类,还会谈到怎样覆盖wxWidgets提供的默认图片和图标.

10.1 wxWidgets中图片相关的类

wxWidgets支持四种和位图相关的类:wxBitmap, wxIcon, wxCursor和wxImage.

wxBitmap是一个平台有关的类,它拥有一个可选的wxMask属性以支持透明绘画.在windows系统上,wxBitmap是通过设备无关位图 (DIBs)实现的,而在GTK+和X11平台上,每个wxBitmap则包含一个GDK的pixmap对象或者X11的pixmap对象.而在Mac平台上,则使用的是PICT.wxBitmap可以和wxImage进行互相转换.

wxIcon用来实现各个平台上的图标,一个图标指的是一个小的透明图片,可以用来代表不同的frame或者对话框窗口.在GTK+, X11和Mac平台上,icon就是一个小的总含有wxMask的wxBitmp,而在windows平台上,wxIcon则是封装了HICON对象.

wxCursor则是一个用来展示鼠标指针的图像,在GTK+平台上是用的GdkCursor,X11和Mac平台上用的是各自的 Cursor,而在windows平台上则使用的是HCURSOR.wxCursor有一个热点的概念(所谓热点指的是图片中用来精确代表指针单击位置的那个点),也总是包含一个遮罩(mask).

wxImage则是四个类中唯一的一个平台无关的实现,它支持24bit位图以及可选的alpha通道.wxImage可以从 wxBitmap类使用wxBitmap::ConvertToImage函数转换而来,也可以从各种各样的图片文件中加载,它所支持的图片格式也是可以通过图片格式处理器来扩展的.它拥有操作其图片上某些bit的能力,因此也可以用来对图片进行一个基本的操作.和wxBitmap不同,wxImage不可以直接被设备上下文wxDC使用,如果要在wxDC上绘图,需要现将wxImage转换成wxBitmap,然后就可以使用wxDC的 DrawBitmap函数进行绘图了.wxImage支持设置一个掩码颜色来实现透明的效果,也支持通过alpha通道实现非常复杂的透明效果.

你可以在这些图片类型之间进行相互转换,尽管某些转换操作是平台相关的.

注意图片类中大量使用引用记数器,因此对图片类进行赋值和拷贝的操作的系统开销是非常小的,不过这也意味着对一个图片的更改可能会影响到别的图片.

所有的图片类都使用下表列出的标准的wxBitmapType标识符来读取或者保存图片数据:

wxBITMAP_TYPE_BMP	Windows位图文件 (BMP).
wxBITMAP_TYPE_BMP_RESOURCE	从windows可执行文件资源部分加载的Windows位图.
wxBITMAP_TYPE_ICO	Windows图标文件(ICO).
wxBITMAP_TYPE_ICO_RESOURCE	从windows可执行文件资源部分加载的Windows图标.
wxBITMAP_TYPE_CUR	Windows光标文件(CUR).
wxBITMAP_TYPE_CUR_RESOURCE	从windows可执行文件资源部分加载的Windows光标.

wxBITMAP_TYPE_XBM	Unix平台上使用的XBM单色图片.
wxBITMAP_TYPE_XBM_DATA	从C++数据中构造的XBM单色位图.
wxBITMAP_TYPE_XPM	XPM格式图片,最好的支持跨平台并且支持编译到应用程序中去的格式.
wxBITMAP_TYPE_XPM_DATA	从C++数据中构造的XPM图片.
wxBITMAP_TYPE_TIF	TIFF格式位图,在大图片中使用比较普遍.
wxBITMAP_TYPE_GIF	GIF格式图片,最多支持256中颜色,支持透明.
wxBITMAP_TYPE_PNG	PNG位图格式, 一个使用广泛的图片格式,支持透明和alpha通道,没有版权问题.
wxBITMAP_TYPE_JPEG	JPEG格式位图, 一个广泛使用的压缩图片格式,支持大图片,不过它的压缩算法是有损耗压缩,因此不适合对图片进行反复加载和压缩.
wxBITMAP_TYPE_PCX	PCX图片格式.
wxBITMAP_TYPE_PICT	Mac PICT位图.
wxBITMAP_TYPE_PICT_RESOURCE	从可执行文件资源部分加载的Mac PICT位图.
wxBITMAP_TYPE_ICON_RESOURCE	仅在Mac OS X平台上有效, 用来加载一个标准的图标(比如wxICON_INFORMATION)或者一个图标资源.
wxBITMAP_TYPE_ANI	Windows动画图标(ANI).
wxBITMAP_TYPE_IFF	IFF位图文件.
wxBITMAP_TYPE_MACCURSOR	Mac光标文件.
wxBITMAP_TYPE_MACCURSOR_RESOURCE	从可执行文件资源部分加载的Mac光标.
wxBITMAP_TYPE_ANY	让加载图片的代码自己确定图片的格式.

10.2 使用wxBitmap编程

你可以使用wxBitmap来作下面的事情:

1. 通过设备上下文将其画在一个窗口上.
2. 在某些类(比如wxBitmapButton, wxStaticBitmap, and wxToolBar)中将其作为一个图片标签.
3. 使用其作为双缓冲区(在将某个图形绘制到屏幕上之前先绘制在一块缓冲区上).

某些平台(特别是windows平台)上限制了系统中bitmap资源的数目,因此如果你需要使用很多的bitmap,你最好使用wxImage类来保存它们,而只在使用的时候将其转化成bitmap.

在讨论怎样创建wxBitmap之前,让我们先来讨论一下几个主要的函数(如下表所示)

wxBitmap	代表一个**bitmap,**可以通过指定宽度和高度**,**或者指定另外一个**bitmap,**或者指定一个**wxImage,XPM**数据**(char**),** 原始数据**(char[]),** 或者一个指定类型的文件名的方式来创建**.**
ConvertToImage	转换成一个wxImage,保留透明部分.
CopyFromIcon	从一个wxIcon创建一个wxBitmap.
Create	从图片数据或者一个给定的大小创建一个bitmap.
GetWidth, GetHeight	返回图片大小.
Getdepth	返回图片颜色深度.
GetMask, SetMask	返回绑定的wxMask对象或者NULL.
GetSubBitmap	将位图其中的某一部分创建为一个新的位图.
LoadFile, SaveFile	从某种支持格式的文件加载或者保存到文件里.
Ok	如果bitmap的数据已经具备则返回True.

创建一个wxBitmap

可以通过下面的几个途径来创建一个wxBitmap对象.

你可以直接通过默认的构造函数创建一个不包含数据的bitmap,不过你几乎不能用这个bitmap作任何事情,除非你调用Create或者LoadFile或者用另外一个bitmap赋值以便使其拥有具体的bitmap数据.

你还可以通过指定宽度和高度的方法创建一个位图,这种情况下创建的位图将被填充以随机的颜色,你需要在其上绘画以便更好的使用它,下面的代码演示了怎样创建一个200x100的图片并且将其的背景刷新为白色.

```
// 使用当前的颜色深度创建一个200x100的位图
wxBitmap bitmap(200, 100,  -1);
// 创建一个内存设备上下文
wxMemoryDC dc;
// 将创建的图片和这个内存设备上下文关联
dc.SelectObject(bitmap);
// 设置背景颜色
dc.SetBackground(*wxWHITE_BRUSH);
// 绘制位图背景
dc.Clear();
// 解除设备上下文和位图的关联
dc.SelectObject(wxNullBitmap);
```

你也可以从一个wxImage对象创建一个位图,并且保留这个image对象的颜色遮罩或者alpha通道:

```
// 加载一幅图像
wxImage image(wxT("image.png"), wxBITMAP_TYPE_PNG);
// 将其转换成bitmap
wxBitmap bitmap(image);
```

通过CopyFromIcon函数可以通过图标文件创建一个bitmap:

```
// 加载一个图标
wxIcon icon(wxT("image.xpm"), wxBITMAP_TYPE_XPM);
// 将其转换成位图
wxBitmap bitmap;
bitmap.CopyFromIcon(icon);
```

或者你可以通过指定类型的方式从一个图片文件直接加载一个位图:

```
// 从文件加载
wxBitmap bitmap(wxT("picture.png", wxBITMAP_TYPE_PNG);
if (!bitmap.Ok())
{
    wxMessageBox(wxT("Sorry, could not load file."));
}
```

wxBitmap可以加载所有的可以被wxImage加载的图片类型,使用的则是各个平台定义的针对特定类型的加载方法.最常用的图片格式比如"PNG, JPG,BMP和XPM"等在各个平台上都支持读取和保存操作,不过你需要确认你的wxWidgets在编译的时候已经打开了对应的支持.

目前支持的图形类型处理函数如下表所示:

wxBMPHandler	用来加载**windows**位图文件.
wxPNGHandler	用来加载PNG类型的文件.这种文件支持透明背景以及alpha通道.
wxJPEGHandler	用来支持JPEG文件
wxGIFHandler	因为版权方面的原因,仅支持GIF格式的加载.
wxPCXHandler	用来支持PCX. wxPCXHandler会自己计算图片中颜色的数目,如果没有超过256色,则采用8bits颜色深度,否则就采用24 bits颜色深度.
wxPNMHandler	用来支持PNM文件格式. 对于加载来说PNM格式可以支持ASCII和raw RGB两种格式.但是保存的时候仅支持raw RGB格式.
wxTIFFHandler	用来支持TIFF.
wxIFFHandler	用来支持IFF格式.
wxXPMHandler	用来支持XPM格式.
wxICOHandler	用来支持windows平台图标文件.
wxCURHandler	用来支持windows平台光标文件.
wxANIHandler	用来支持windows平台动画光标文件.

在Mac OS X平台上,还可以通过指定wxBITMAP_TYPE_PICT_RESOURCE来加载一个PICT资源.

如果你希望在不同的平台上从不同的位置加载图片,你可以使用wxBITMAP宏,如下所示:

```
#if !defined(__WXMSW__) && !defined(__WXPM__)
#include "picture.xpm"
#endif
wxBitmap bitmap(wxBITMAP(picture));
```

这将使得程序在windows和OS/2平台上从资源文件中加载图片,而在别的平台上,则从一个picture_xpm变量中加载xpm格式的图片,因为XPM在所有的平台上都支持,所以这种使用方法并不常见.

设置一个wxMask

每个wxBitmap对象都可以指定一个wxMask,所谓wxMask指的是一个单色图片用来指示原图中的透明区域.如果你要加载的图片中包含透明区域信息(比如XPM,PNG或者GIF格式),那么wxMask将被自动创建,另外你也可以通过代码创建一个wxMask然后调用 SetMask函数将其和对应的wxBitmap对象相关连.你还可以从wxBitmap对象创建一个wxMask,或者通过给一个wxBitmap对象指定一种透明颜色来创建一个wxMask对象.

下面的代码创建了一个拥有透明色的灰阶位图mainBitmap,它的大小是32x32象素,原始图形从imageBits数据创建,遮罩图形从maskBits创建,遮罩中1代表不透明,0代表透明颜色.

```
static char imageBits[] = { 255, 255, 255, 255, 31,
  255, 255, 255, 31, 255, 255, 255, 31, 255, 255, 255,
  31, 255, 255, 255, 31, 255, 255, 255, 31, 255, 255,
  255, 31, 255, 255, 255, 31, 255, 255, 255, 25, 243,
  255, 255, 19, 249, 255, 255, 7, 252, 255, 255, 15, 254,
  255, 255, 31, 255, 255, 255, 191, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
  255 };
static char maskBits[] = { 240, 1, 0, 0, 240, 1,
  0, 0, 240, 1, 0, 0, 240, 1, 0, 0, 240, 1, 0, 0, 240, 1,
  0, 0, 240, 1, 0, 0, 240, 1, 0, 0, 255, 31, 0, 0, 255,
  31, 0, 0, 254, 15, 0, 0, 252, 7, 0, 0, 248, 3, 0, 0,
  240, 1, 0, 0, 224, 0, 0, 0, 64, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0 };
wxBitmap mainBitmap(imageBits, 32, 32);
wxBitmap maskBitmap(maskBits, 32, 32);
mainBitmap.SetMask(new wxMask(maskBitmap));
```

XPM图形格式

在使用小的需要支持透明的图片(比如工具栏上的小图片或者notebook以及树状控件上的小图片)的时候,wxWidgets的程序员通常偏爱使用XPM格式,它的最大的特点是采用C/C++语言的语法,既可以被程序动态加载,也可以直接编译到可执行代码中去,下面是一个例子:

```
// 你也可以用 #include "open.xpm"
static char *open_xpm[] = {
/* 列数 行数 颜色个数 每个象素的字符个数 */
"16 15 5 1",
"  c None",
". c Black",
"X c Yellow",
"o c Gray100",
"O c #bfbf00",
/* 象素 */
"                ",
"          ...   ",
"         .   . .",
"              ..",
"  ...        ...",
" .XoX.......    ",
" .oXoXoXoXo.    ",
" .XoXoXoXoX.    ",
" .oXoX..........",
" .XoX.OOOOOOOOO.",
" .oo.OOOOOOOOO. ",
" .X.OOOOOOOOO.  ",
" ..OOOOOOOOO.   ",
" ...........    ",
"                "
};
wxBitmap bitmap(open_xpm);
```

正如你看到的那样,XPM是使用字符编码的.在图片数据前面,有一个调色板区域,使用字符和颜色对应的方法,颜色既可以用标准标识符表示,也可以用16进制的RGB值表示,使用关键字None来表示透明区域.尽管在windows系统上,XPM并不被大多数的图形处理工具支持,不过你还是可以通过一些工具把PNG格式转换成XPM格式,比如DialogBlocks自带的ImageBlocks工具,或者你可以直接使用 wxWidgets编写一个你自己的转换工具.

使用位图绘画

使用位图绘画的方式有两种,你可以使用内存设备上下文绑定一个位图,然后使用wxDC::Blit函数,也可以直接使用wxDC:: DrawBitmap函数,前者允许你使用位图的一部分进行绘制.在两种方式下,如果这个图片支持透明或者alpha通道,你都可以通过将最后一个参数指定为True或者False来打开或者关闭透明支持.

这两种方法的用法如下:

```
// Draw a bitmap using a wxMemoryDC
wxMemoryDC memDC;
memDC.SelectObject(bitmap);
// Draw the bitmap at 100, 100 on the destination DC
destDC.Blit(100, 100,                         // Draw at (100, 100)
    bitmap.GetWidth(), bitmap.GetHeight(),    // Draw full bitmap
    & memDC,                                  // Draw from memDC
    0, 0,                                     // Draw from bitmap origin
    wxCOPY,                                   // Logical operation
    true);                                    // Take mask into account
memDC.SelectObject(wxNullBitmap);
// Alternative method: use DrawBitmap
destDC.DrawBitmap(bitmap, 100, 100, true);
```

第五章,"绘画和打印"中对使用bitmap绘画有更详细的描述.

打包位图资源

如果你曾是一个windows平台的程序员,你可能习惯从可执行文件的资源部分加载一幅图片,当然在wxWidgets中也可以这样作, 你只需要指定一个资源名称一个资源类型wxBITMAP_TYPE_BMP_RESOURCE,不过这种作法是平台相关的.你可能更倾向于使用另外一种平台无关的解决方案.

一个可移植的方法是,你可以将你用到的所有数据文件,包括HTML网页,图片或者别的任何类型的文件压缩在一个zip文件里,然后你可以用wxWidgets提供的虚拟文件系统对加载这个zip文件的其中任何一个或几个文件,如下面的代码所示:

```
// 创建一个文件系统
wxFileSystem*fileSystem = new wxFileSystem;
wxString archiveURL(wxT("myapp.bin"));
wxString filename(wxT("myimage.png"));
wxBitmapType bitmapType = wxBITMAP_TYPE_PNG;
// 创建一个URL
wxString combinedURL(archiveURL + wxString(wxT("#zip:")) + filename);
wxImage image;
wxBitmap bitmap;
// 打开压缩包中的对应文件
wxFSFile* file = fileSystem->OpenFile(combinedURL);
if (file)
{
    wxInputStream* stream = file->GetStream();

    // Load and convert to a bitmap
    if (image.LoadFile(* stream, bitmapType))
        bitmap = wxBitmap(image);

    delete file;
}
delete fileSystem;
if (bitmap.Ok())
{
    ...
}
```

更多关于虚拟文件系统的信息请参考第14章:文件和流操作.

10.3 使用wxIcon编程

一个wxIcon代表一个小的位图,它总有一个透明遮罩,它的用途包括:

- 设置frame窗口或者对话框的图标
- 通过wxImageList类给wxTreeCtrl, wxListCtrl或者wxNotebook提供图标 (更多信息请参考最后一章)
- 使用wxDC::DrawIcon函数在设备上下文中绘制一个图标

下表列出了图标类的主要成员函数

wxIcon	图标类可以通过指定另外一个图标类的方式,指定**XPM**数据**(char**)**的方式, 原始数据**(char[])**的方式,或者文件名及文件类型的方式创建**.**
CopyFromBitmap	从wxBitmap类创建一个图标.
GetWidth, GetHeight	返回图标的大小.
Getdepth	返回图标的颜色深度.
LoadFile	从文件加载图标.
Ok	在图标数据已经具备的时候返回True.

创建一个wxIcon

wxIcon可以使用XPM数据创建,或者从一个wxBitmap对象中创建,或者从文件(比如一个Xpm文件)中读取.wxWidgets也提供了类似于前一小节提到的wxBITMAP类似的宏,用来从一个平台相关的资源中获取图标.

在windows平台上,LoadFile以及同等性质的操作可以使用的文件类型包括BMP图片和ICO文件,如果你要从其它图片格式中创建图标,可以先将其读入一个wxBitmap对象中,然后再将其转换为一个图标.

而在Mac OSX和 Unix/Linux的GTK+版本中,wxIcon可以识别的图片类型和wxBitmap可以识别的图片类型是一样的.

下面代码演示了创建一个wxIcon对象的几种方法:

```
// 方法1: 从XPM数据创建
#include "icon1.xpm"
wxIcon icon1(icon1_xpm);
// 方法2: 从一个ICO资源中创建(Window and OS/2 only)
wxIcon icon2(wxT("icon2"));
// 方法3: 从一个图片文件中 (Windows and OS/2 only)
// 如果你的图片包含多个图标你可以指定单个图标的宽度
wxIcon icon3(wxT("icon3.ico"), wxBITMAP_TYPE_ICO, 16, 16);
// 方法4: 从位图创建
wxIcon icon4;
wxBitmap bitmap(wxT("icon4.png"), wxBITMAP_TYPE_PNG);
icon4.CopyFromBitmap(bitmap);
```

使用wxIcon

下面的代码演示了wxIcon的三种使用方法:设置窗口图标,增加到一个图片列表或者绘制在某个设备上下文上

```
#include "myicon.xpm"
wxIcon icon(myicon_xpm);
// 1: 设置窗口图标
frame->SetIcon(icon);
// 2: 增加到wxImageList
wxImageList* imageList = new wxImageList(16, 16);
imageList->Add(icon);
// 3: 在(10, 10)的位置绘制
wxClientDC dc(window);
dc.DrawIcon(icon, 10, 10);
```

将某个图标绑定到应用程序

将某个图标绑定到应用程序,以便系统可以显示这个图标在合适的位置使得用户可以通过点击图标的方式打开应用程序,这个工作wxWidgets是做不到的.这是极少的你需要在不同的平台使用不同的技术的领域中的一个.

在windows平台上,你需要在makefile中增加一个资源文件(扩展名是.rc),并且在这个资源文件中指定一个图标区域,如下所示:

```
aardvarkpro ICON aardvarkpro.ico
#include "wx/msw/wx.rc"
```

在这里, aardvarkpro.ico就是这个和应用程序绑定的图标的名称,它可以有多种分辨率和颜色深度(典型的大小包括48x48,32x32和 16x16).当windows的资源管理器需要显示某个图标的时候,它将使用子母顺序排在第一个的那个图标,因此你最好给确定要作为应用程序图标的那个图标的名称前面加几个a子母以便按照子母顺序它排在前面,否则你的应用程序可能绑定的是你不期望的图标.

在Mac系统上,你需要准备一个应用程序包,其中包含一些ICNS文件.参考第20章"让你的程序更完美",来获得关于程序包更多的信息,其中的主要文件Info.plist文件看上去应该象下面的额样子:

```
<key>CFBundleDocumentTypes</key>
<array>
     <dict>
           <key>CFBundleTypeExtensions</key>
           <array>
                    <string>pjd</string>
             </array>
             <key>CFBundleTypeIconFile</key>
             <string>dialogblocks-doc.icns</string>
             <key>CFBundleTypeName</key>
             <string>pjdfile</string>
             <key>CFBundleTypeRole</key>
             <string>Editor</string>
       </dict>
</array>
<key>CFBundleIconFile</key>
<string>dialogblocks-app.icns</string>
...
```

应用程序图标和应用程序相关的文档类型图标是由CFBundleIconFile和CFBundleTypeIconFile属性指定的.你可以直接用Apple提供图标编辑器编辑ICNS文件,不过如果你希望所有的平台使用同样的图标,你最好现用PNG图片创建各种大小的图标,然后再将它粘贴到各个平台上的图标编辑器中,要确保PNG使用的透明遮罩颜色和各个工具使用的透明颜色相一致.

而在linux平台上,Gnome桌面系统和KDE桌面系统则各自拥有自己的图标提供体系,我们将在第20章进行简要的描述.

10.4 使用wxCursor编程

光标用来指示鼠标指针当前的位置.你可以给某个窗口指定不同的光标以便提示用户这个窗口期待某种类型的鼠标操作.和图标一样,光标也是一种始终带有透明遮罩的小图片,可以使用一般的构造函数或者是平台相关的构造函数来创建.其中的一些构造函数还需要相对于整个图片的左上角指定一个热点位置,当鼠标点击的时候,热点所在的位置将作为鼠标点击的位置.

下表列举了光标相关的函数

wxCursor	光标可以从**wxImage**对象**,**二进制数据**,**系统定义的光标标识符以及光标文件来创建**.**
Ok	如果光标数据已经具备,则返回True.

创建一个光标

创建光标最简单的方法是通过系统提供的光标标识符,如下面的例子所示:

wxCursor cursor(wxCURSOR_WAIT);

下表列出了目前支持的光标标识符和它们的光标的样子(依照平台的不同会有些变化)

wxCURSOR_ARROW		标准光标.
wxCURSOR_RIGHT_ARROW		标准反向光标.
wxCURSOR_BLANK	透明光标.	
wxCURSOR_BULLSEYE		近视眼.
wxCURSOR_CROSS		十字.
wxCURSOR_HAND		手.
wxCURSOR_IBEAM		I字光标.
wxCURSOR_LEFT_BUTTON		按左键(GTK+ only).
wxCURSOR_MAGNIFIER		放大镜.
wxCURSOR_MIDDLE_BUTTON		按中键(译者注:原书图片有误) (GTK+ only).
wxCURSOR_NO_ENTRY		禁止通行.
wxCURSOR_PAINT_BRUSH		画刷.
wxCURSOR_PENCIL		铅笔.
wxCURSOR_POINT_LEFT		向左.
wxCURSOR_POINT_RIGHT		向右.
wxCURSOR_QUESTION_ARROW		带问号的箭头.
wxCURSOR_RIGHT_BUTTON		按右键(译者注:图片有误) (GTK+ only).
wxCURSOR_SIZENESW		东北到西南伸缩.
wxCURSOR_SIZENS		南北伸缩.
wxCURSOR_SIZENWSE		西北到东南伸缩.
wxCURSOR_SIZEWE		东西伸缩.
wxCURSOR_SIZING		一般伸缩.
wxCURSOR_SPRAYCAN		画刷.
wxCURSOR_WAIT		等待.
wxCURSOR_WATCH		查看.
wxCURSOR_ARROWWAIT		后台忙.

你还可以使用预定义光标指针wxSTANDARD_CURSOR, wxHOURGLASS_CURSOR和wxCROSS_CURSOR.

另外在windows和Mac平台上还可以从对应的资源文件中加载光标:

```
//windows平台
wxCursor cursor(wxT("cursor_resource"), wxBITMAP_TYPE_CUR_RESOURCE,
                hotSpotX, hotSpotY);
// Mac平台
wxCursor cursor(wxT("cursor_resource"), wxBITMAP_TYPE_MACCUR_RESOURCE);
```

你还可以通过wxImage对象创建光标,而"热点"则要通过wxImage::SetOptionInt函数设置.之所以要设置热点, 是因为很多光标不太适合使用默认的左上角作为热点,比如对于十字光标来说,你可能希望将其十字交叉的地方作为热点.下面的代码演示了怎样从一个PNG文件中产生设置了热点的光标:

```
// 用wxImage创建光标
wxImage image(wxT("cursor.png"), wxBITMAP_TYPE_PNG);
image.SetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_X, 5);
image.SetOptionInt(wxIMAGE_OPTION_CUR_HOTSPOT_Y, 5);
wxCursor cursor(image);
```

使用wxCursor

每个窗口都可以设置一个对应的光标,这个光标在鼠标进入这个窗口的时候显示,如果一个窗口没有设置光标,其父窗口的光标将被显示,如果所有的父窗口都没有设置光标,则系统默认光标被显示:

使用下面的代码给窗口设置一个光标:

```
window->SetCursor(wxCursor(wxCURSOR_WAIT));
```

使用wxSetCursorEvent

在windows系统或者是Mac OS X系统上,有一些小地方我们需要注意一下.举个例子,如果你自己实现了一个容器类,比方说是一个分割窗口,并且给它设置了一个特殊的光标(比如说wxCURSOR_WE用来表明某个分割条是可以被拉动的),然后你在这个分割窗口中放置了两个子窗口,如果你没有给这两个子窗口设置光标的话,当光标在子窗口上移动时,它们可能会不恰当的显示其父窗口,那个wxCURSOR_WE光标.而本来你是希望只有在鼠标移动到分割条上的时候才显示的.

要告诉wxWidgets某个光标只应该在某种情况下被显示,你可以增加一个wxSetCursorEvent事件的处理函数,这个事件在Windows和Mac平台上,当需要设置光标的时候(通常是鼠标在窗口间移动的时候)被产生.在这个事件处理函数中可以调用 wxSetCursorEvent::SetCursor来设置一个特殊的光标.如下所示:

```
BEGIN_EVENT_TABLE(wxSplitterWindow, wxWindow)
    EVT_SET_CURSOR(wxSplitterWindow::OnSetCursor)
END_EVENT_TABLE()
// 指示光标只应该被设置给分割条
void wxSplitterWindow::OnSetCursor(wxSetCursorEvent& event)
{
    if ( SashHitTest(event.GetX(), event.GetY(), 0) )
    {
        // 使用默认的处理
        event.Skip();
    }
    //else:什么也不作,换句话说,不调用Skip.则事件表不会被继续搜索
}
```

在这个例子中,当鼠标指针移过分割条的时候,SashHitTest函数返回True,因此Skip函数被调用,事件表调用失败,这和没有定义这个事件表的效果是一样的,导致wxWidgets象往常一样显示指定给窗口的光标(wxCURSOR_WE).而如果SashHitTest函数返回False,则表明光标是在子窗口上移动,这时候应该不显示我们指定的光标,因此我们不调用Skip函数,让事件表匹配成功,则事件表将不会在继续匹配,这将使得wxWidgets认为这个窗口没有被指定光标,因此.在这种情况下,即使子窗口自己没有光标(象wxTextCtrl这种控件,一般系统会指定一个它自己的光标,不过wxWidgets对这个是不感知的),也将不会使用我们指定给父窗口的光标.

10.5 使用wxImage编程

你可以使用wxImage对图形进行一些平台无关的调整,或者将其作为图片加载和保存的中间步骤.图片在wxImage中是按照每一个象素使用一个分别代表红色,绿色和蓝色的字节的格式保存的,如果图片包含alpha通道,则还会占用额外的一个字节.

wxImage主要的函数如下:

wxImage	**wxImage的创建方法包括:指定宽度和高度, 从另外一幅图片创建, 使用XPM数据, 图片元数据(char[]) 和可选的alpha通道数据,文件名及其类型,以及通过输入流等多种方式创建.**
ConvertAlphaToMask	将alpla通道(如果有的话)转换成一个透明遮罩.
ConvertToMono	转换成一个黑白图片.
Copy	返回一个不使用引用记数器的完全一样的拷贝.
Create	创建一个指定大小的图片,可选的参数指明是否初始化图片数据.
Destroy	如果没有人再使用的话,释放内部数据.
GeTData, SetData	获取和设置内部数据指针(unsigned char*).
GetImageCount	返回一个文件或者流中的图片个数.
GetOption, GetOptionInt, SetOption, HasOption	获取, 设置和测试某个选项是否设置.
GetSubImage	将图片的一部分返回为一个新的图像.
GetWidth, GetHeight	返回图片大小.
Getred, GetGreen, GetBlue, SetRGB, GetAlpha, SetAlpha	获得和指定某个象素的RGB以及Alpha通道的值.
HasMask, GetMaskRed, GetMaskGreen, GetMaskBlue, SetMaskColour	用来测试图像是否有一个遮罩,以及遮罩颜色的RGB值或者整个颜色的值.
LoadFile, SaveFile	各种图片格式文件的读取和保存操作.
Mirror	在各种方向上产生镜像,返回一个新图片.
Ok	判断图片是否已初始化.
Paste	将某个图片粘贴在这个图片的指定位置.
Rotate, Rotate90	旋转图片,返回一个新图片.
SetMaskFromImage	通过指定的图片和透明颜色产生一个遮罩并且设置这个遮罩.
Scale, Rescale	缩放产生一个新图片或者缩放本图片.

加载和保存图像

wxImage可以读取和保存各种各样的图片格式,并且使用图像处理过程来增加扩展的能力.其它的图像类(比如wxBitmap)在某个平台不具备处理某种图形格式的能力的时候,也通常使用的都是wxImage的图象处理过程来加载特定格式的图形.

本章第二小节中展示了wxWidgets支持的各种图形处理过程.其中wxBMPHandler是默认支持的,而要支持其它的图形格式处理,就需要使用 wxImage::AddHandler函数增加对应的图形处理过程或者使用wxInitAllImageHandlers增加所有支持的图形处理过程.

如果你只需要特定的图形格式支持,可以在OnInit函数中使用类似下面的代码:

```
#include "wx/image.h"
wxImage::AddHandler( new wxPNGHandler );
wxImage::AddHandler( new wxJPEGHandler );
wxImage::AddHandler( new wxGIFHandler );
wxImage::AddHandler( new wxXPMHandler );
```

或者,你可以简单的调用:

```
wxInitAllImageHandlers();
```

下面演示了几种从文件或者流读取图片的方式,注意在实际使用过程中,最好使用绝对路径以避免依赖于当前路径的设置:

```
// 使用构造函数指定类型来读取图像
wxImage image(wxT("image.png"), wxBITMAP_TYPE_PNG);
if (image.Ok())
{
    ...
}
// 不指定图像类型一般也能正常工作
wxImage image(wxT("image.png"));
// 使用两步法创建图像
wxImage image;
if (image.LoadFile(wxT("image.png")))
{
    ...
}
// 如果一个文件包含两副图片Two-step loading with an index into a multi-image file:
// 下面演示选择第2副加载
wxImage image;
int imageCount = wxImage::GetImageCount(wxT("image.tif"));
if (imageCount &gt; 2)
    image.LoadFile(wxT("image.tif"), wxBITMAP_TYPE_TIFF, 2);
// 从文件流加载图片
wxFileInputStream stream(wxT("image.tif"));
wxImage image;
image.LoadFile(stream, wxBITMAP_TYPE_TIF);
// 保存到一个文件
image.SaveFile(wxT("image.png")), wxBITMAP_TYPE_PNG);
// 保存到一个流
wxFileOutputStream stream(wxT("image.tif"));
image.SaveFile(stream, wxBITMAP_TYPE_TIF);
```

除了XPM和PCX格式以外,其它的图片格式都将以24位颜色深度保存(译者注:GIF格式因为版权方面的原因不支持保存到文件),这两种格式的图形处理过程将会计算实际的颜色个数从而选择相应的颜色深度.JPEG格式还拥有一个质量选项可供设置.它的值的范围为从0到100,0代表最低的图片质量和最高的压缩比,100则代表最高的图片质量和最低的压缩比.如下所示:

```
// 设置一个合理的质量压缩比
image.SetOption(wxIMAGE_OPTION_QUALITY, 80);
image.SaveFile(wxT("picture.jpg"), wxBITMAP_TYPE_JPEG);
```

另外如果以XPM格式保存到流输出中的时候,需要使用wxImage::SetOption函数设置一个名称否则,处理函数不知道该用什么名称命名对应的C变量.

```
// 保存XPM到流格式
image.SetOption(wxIMAGE_OPTION_FILENAME, wxT("myimage"));
image.SaveFile(stream, wxBITMAP_TYPE_XPM);
```

注意处理函数会自动在你设置的名称后增加"_xpm".

透明

有两种方式设置一个wxImage为透明的图像:使用颜色遮罩或者alpha通道.一种颜色可以被指定为透明颜色,通过这种方法在将wxImage转换成wxBitmap的时候可以很容易的制作一个透明遮罩.

wxImage也支持alpha通道数据,在每一个象素的RGB颜色之外来由另外一个字节用来指示alpha通道的值,0代表完全透明,255则代表完全不透明.中间的值代表半透明.

不是所有的图片都用有alpha通道数据的,因此在使用GetAlpha函数之前,应该使用HasAlpha函数来判断图像是否拥有 alpha通道数据.到目前为止,只有PNG文件或者调用SetAlpha设置了alpha通道的图像才拥有alpha通道数据.保存一个带有alpha 通道的图像目前还不被支持.绘制一个拥有alpha通道的方法是先将其转换成wxBitmap然后使用wxDC::DrawBitmap或者wxDC::Blit函数.

下面的代码演示了怎样使用颜色掩码创建一个透明的wxImage,它是蓝色的,拥有一个透明的矩形区域:

```
// 创建一个有颜色掩码的wxBitmap
// 首先,在这个wxBitmap上绘画
wxBitmap bitmap(400, 400);
wxMemoryDC dc;
dc.SelectObject(bitmap);
dc.SetBackground(*wxBLUE_BRUSH);
dc.Clear();
dc.SetPen(*wxRED_PEN);
dc.SetBrush(*wxRED_BRUSH);
dc.DrawRectangle(50, 50, 200, 200);
dc.SelectObject(wxNullBitmap);
// 将其转换成wxImage
wxImage image = bitmap.ConvertToImage();
// 设置掩码颜色
image.SetMaskColour(255, 0, 0);
```

在下面的例子中,使用从一个图片创建颜色遮罩的方式,其中image.bmp是原始图像,而mask.bmp则是一个掩码图像,在后者中所有透明的部分都是黑色显示的.

```
// 加载一副图片和它的掩码遮罩
wxImage image(wxT("image.bmp"), wxBITMAP_TYPE_BMP);
wxImage maskImage(wxT("mask.bmp"), wxBITMAP_TYPE_BMP);
// 从后者创建一个遮罩并且设置给前者.
image.SetMaskFromImage(maskImage, 0, 0, 0);
```

如果你加载的图片本身含有透明颜色,你可以检测并且直接创建遮罩:

```
// 加载透明图片
wxImage image(wxT("image.png"), wxBITMAP_TYPE_PNG);
// 获取掩码
if (image.HasMask())
{
    wxColour maskColour(image.GetMaskRed(),
    image.GetMaskGreen(),
    image.GetMaskBlue());
}
```

变形

wxImage支持缩放,旋转以及镜像等多种变形方式,下面各举一些例子:

```
// 把原始图片缩放到200x200,并保存在新的图片里
// 原图保持不变.
wxImage image2 = image1.Scale(200, 200);
// 将原图缩放到200x200
image1.Rescale(200, 200);
// 旋转固定角度产生新图片.
// 原图片保持不变.
wxImage image2 = image1.Rotate(0.5);
// 顺时针旋转90度产生新图片.
// 原图保持不变.
wxImage image2 = image1.Rotate90(true);
// 水平镜像产生新图片.
// 原图保持不变.
wxImage image2 = image1.Mirror(true);
```

颜色消减

如果你想对某个图像的颜色进行消减,你可以使用wxQuantize类的一些静态函数,其中最有趣的函数Quantize的参数为一个输入图片,一个输出图片,一个可选的wxPalette指针用来存放经过消减的颜色,以及一个你希望保留的颜色个数,你也可以传递一个**unsigned char**变量来获取一个8-bit颜色深度的输出图像.最后的一个参数style(类型)用来对返回的图像进行一些更深入的控制,详情请参考 wxWidgets的手册.

下面的代码演示了怎样将一幅图片的颜色消减到最多256色:

```
#include "wx/image.h"
#include "wx/quantize.h"
wxImage image(wxT("image.png"));
int maxColorCount = 256;
int colors = image.CountColours();
wxPalette* palette = NULL;
if (colors &gt; maxColorCount )
{
    wxImage reducedImage;
    if (wxQuantize::Quantize(image, reducedImage,
                              & palette, maxColorCount))
    {
        colors = reducedImage.CountColours();
        image = reducedImage;
    }
}
```

一个wxImage可以设置一个wxPalette,例如加载GIF文件的时候. 然后,图片内部仍然是以RGB的方式存储数据的,调色板仅代表图片加载时候的颜色隐射关系.调色板的另外一个用途是某些图片处理函数用它来将图片保存为低颜色深度的图片,例如windows的BMP图片处理过程将检测是否设置了wxBMP_8BPP_PALETTE标记,如果设置了,则将使用调色板.而如果设置了wxBMP_8BPP标记(而不是wxBMP_8BPP_PALETTE),它将使用自己的算法进行颜色消减.另外某些图片处理过程自己也进行颜色消减,比如PCX的处理过程,除非它认为剩余的颜色个数已经足够低了,否则它将对图片的颜色进行消减.

关于调色板更多的信息请参考第5章的"调色板"小节.

直接操作wxImage 的元数据

你可以直接通过GetData函数访问wxImage的元数据以便以比GeTRed, GetBlue, GetGreen和SetRGB更快的方式对其进行操作,下面举了一个使用这种方法将一个图片转换成灰度图片的方法:

```
void wxImage::ConvertToGrayScale(wxImage& image)
{
    double red2Gray   = 0.297;
    double green2Gray = 0.589;
    double blue2Gray  = 0.114;
    int w = image.GetWidth(), h = image.GetHeight();
    unsigned char *data = image.GetData();
    int x,y;
    for (y = 0; y &lt; h; y++)
        for (x = 0; x &lt; w; x++)
        {
            long pos = (y * w + x) * 3;
            char g = (char) (data[pos]*red2Gray +
                             data[pos+1]*green2Gray +
                             data[pos+2]*blue2Gray);
            data[pos] = data[pos+1] = data[pos+2] = g;
        }
}
```

10.6 图片列表和图标集

有时候,使用一组图片是非常方便的.这时候,你可以直接在你的代码中使用wxImageList,也可以和wxWidgets提供的一些控件一起使用 wxImageList,wxNotebook,wxtreeCtrl和wxListCtrl都需要wxImageList来管理它们所需要使用的图标. 你也可使用wxImageList中的某个单独的图片在设备上下文上绘画.

创建一个wxImageList需要的参数包括单个图片的宽度和高度,一个bool值来指定是否需要指定图片遮罩,以及这个图片列表的初始大小(主要是为了内部优化代码),然后一个一个的增加wxBitmap对象或者wxIcon对象.wxImageList不能直接使用wxImage 对象,你需要先将其转换为wxBitmap对象.wxImageList::Add函数返回一个整数的索引用来代表这个刚增加的图片,在Add函数成功返回以后,你就可以释放原始图片了,wxImageList已经在内部创建了一个这个图片的拷贝.

下面是创建wxImageList以及在其中增加图片的一些例子:

```
// 创建一个wxImageList
wxImageList *imageList = new wxImageList(16, 16, true, 1);
// 增加一个透明的PNG文件
wxBitmap bitmap1(wxT("image.png"), wxBITMAP_TYPE_PNG);
imageList->Add(bitmap1);
// 增加一个透明的来自别的bitmap的图片
wxBitmap bitmap2(wxT("image.bmp"), wxBITMAP_TYPE_BMP);
wxBitmap maskBitmap(wxT("mask.bmp"), wxBITMAP_TYPE_BMP);
imageList->Add(bitmap2, maskBitmap);
// 增加一个指定透明颜色的透明图片
wxBitmap bitmap3(wxT("image.bmp"), wxBITMAP_TYPE_BMP);
imageList->Add(bitmap3, *wxRED);
// 增加一个图标
#include "folder.xpm"
wxIcon icon(folder_xpm);
imageList->Add(icon);
```

你可以直接把wxImageList中的图片绘制在设备上下文上,通过指定wxIMAGELIST_DRAW_TRANSPARENT类型来指示绘制透明图片,你还可以指定的类型包括wxIMAGELIST_DRAW_NORMAL, wxIMAGELIST_DRAW_SELECTED或者wxIMAGELIST_DRAW_FOCUSED,用来表征图片的状态,如下所示:

```
// 绘制列表中所有的图片
wxClientDC dc(window);
size_t i;
for (i = 0; i &lt; imageList->GetImageCount(); i++)
{
    imageList->Draw(i, dc, i*16, 0, wxIMAGELIST_DRAW_NORMAL|
                                    wxIMAGELIST_DRAW_TRANSPARENT);
}
```

要把图片列表和notebook的TAB页面绑定在一起,你需要创建一个包含大小为16x16的图片的列表,然后调用 wxNotebook::SetImageList或者wxNotebook::AssignImageList将其和某个wxNotebook绑定,这两个函数的区别在于,前者在wxNotebook释放的时候不释放列表,而后者在自己被释放的时候,会同时释放图片列表.指定完图片列表以后,你就可以给某个页面指定图标索引以便在页面标签上显示图标了,下面的代码演示了这个过程:

```
//创建一个wxImageList
wxImageList *imageList = new wxImageList(16, 16, true, 1);
// 增加一些图标
wxBitmap bitmap1(wxT("folder.png"), wxBITMAP_TYPE_PNG);
wxBitmap bitmap2(wxT("file.png"), wxBITMAP_TYPE_PNG);
int folderIndex = imageList->Add(bitmap1);
int fileIndex = imageList->Add(bitmap2);
// 创建一个拥有两个页面的notebook
wxNotebook* notebook = new wxNotebook(parent, wxID_ANY);
wxPanel* page1 = new wxPanel(notebook, wxID_ANY);
wxPanel* page2 = new wxPanel(notebook, wxID_ANY);
// 绑定图片列表
notebook->AssignImageList(imageList);
// Add the pages, with icons
notebook->AddPage(page1, wxT("Folder options"), true, folderIndex);
notebook->AddPage(page2, wxT("File options"), false, fileIndex);
```

wxtreeCtrl和wxListCtrl的使用方法和上面介绍的非常相似,也包含类似的两种绑定方法.

如果你拥有很多图标,有时候很难通过索引来对应到具体的图标,你可能想编写一个类以便通过字符串来找到某个图片索引.下面演示了基于这个目的的一个简单的实现:

```
#include "wx/hashmap.h"
WX_DECLARE_STRING_HASH_MAP(int, IconNameToIndexHashMap);
// 通过名字引用图片的类
class IconNameToIndex
{
public:
    IconNameToIndex() {}
    // 在图片列表中增加一个已经命名的图片
    void Add(wxImageList* list, const wxBitmap& bitmap,
        const wxString& name) {
        m_hashMap[name] = list->Add(bitmap);
    }
    // 在图片列表中增加一个已命名的图标
    void Add(wxImageList* list, const wxIcon& icon,
        const wxString& name) {
        m_hashMap[name] = list->Add(icon);
    }
    // 通过名称找到索引
    int Find(const wxString& name) { return m_hashMap[name]; }
private:
    IconNameToIndexHashMap m_hashMap;
};
```

wxIconBundle类同样也是一个图片列表,不过这个类的目的是为了将多个不同分辨率的图标保存在一个类中而不是多个类中,以便系统在合适的时候根据不同的使用目的选择一个特定的图标.比如,在资源管理器中的图标通常比在主窗口标题栏上显示的图标要大的多.下面的例子演示了其用法:

```
// 创建一个只有单个16x16图标的图片集
#include "file16x16.xpm"
wxIconBundle iconBundle(wxIcon(file16x16_xpm));
// 在图片集中增加一个32x32的图片
iconBundle.Add(wxIcon(wxT("file32x32.png"), wxBITMAP_TYPE_PNG));
// 从一个包含多个图片的文件中创建一个图片集
wxIconBundle iconBundle2(wxT("multi-icons.tif"), wxBITMAP_TYPE_TIF);
// 从图片集中获取指定大小的图片,如果找不到则继续寻找
// wxSYS_ICON_X, wxSYS_ICON_Y大小的图片
wxIcon icon = iconBundle.GetIcon(wxSize(16,16));
// 将图片集指定给某个主窗口
wxFrame* frame = new wxFrame(parent, wxID_ANY);
frame->SetIcons(iconBundle);
```

在windows系统上,SetIcons函数期待一个包含16x16和32x32大小的图标的图标集.

10.7 自定义wxWidgets提供的小图片

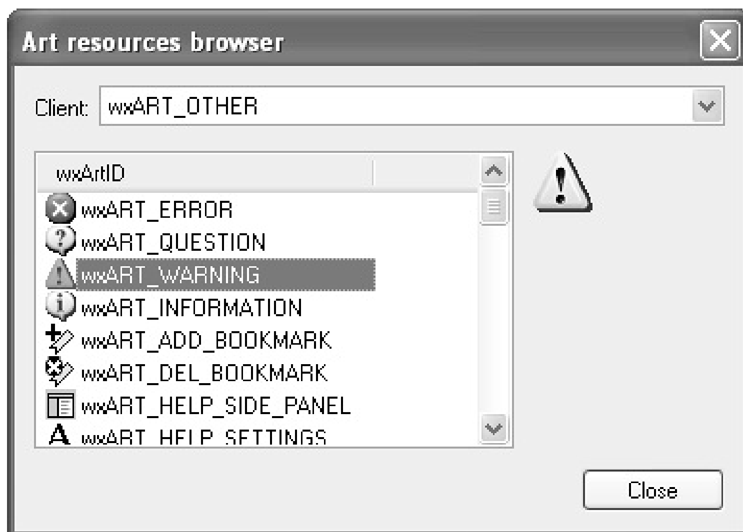
wxArtProvider这个类允许你更改wxWidgets默认提供的那些小图片,比如wxWidgets HTML帮助阅读器中或者默认的Log对话框中使用的图片.

wxWidgets提供了一个标准的wxArtProvider对象,并且体系内的一些需要使用图标和小图片的地方都调用了这个类的wxArtProvider::GetBitmap和wxArtProvider::GetIcon函数.

小图片是由两个标识符决定的:主标识符(wxArtID)和客户区标识符(wxArtClient).其中客户区标识符只在同一个主标识符在不同的窗口中需要不同的图片的时候才使用,比如,wxHTML帮助窗口使用的图标使用下面的代码取得的:

```
wxBitmap bmp = wxArtProvider::GetBitmap(wxART_GO_BACK,wxART_TOOLBAR);
```

如果你想浏览所有wxWidgets提供的小图片以及它们的标识符,你可以编译和运行wxWidgets自带的samples/artprov中的例子,它的外观如下图所示:



要替换wxWidgets提供的这些小图片,你需要实现一个wxArtProvider的派生类,重载其中的CreateBitmap函数,然后在 OnInit函数中调用wxArtProvider::PushProvider以便让wxWidgets知道.下面的这个例子替换了wxHTML帮助窗口中的大部分默认的图标:

```
// 新的图标
#include "bitmaps/helpbook.xpm"
#include "bitmaps/helppage.xpm"
#include "bitmaps/helpback.xpm"
#include "bitmaps/helpdown.xpm"
#include "bitmaps/helpforward.xpm"
#include "bitmaps/helpoptions.xpm"
#include "bitmaps/helpsidepanel.xpm"
#include "bitmaps/helpup.xpm"
#include "bitmaps/helpuplevel.xpm"
#include "bitmaps/helpicon.xpm"
#include "wx/artprov.h"
class MyArtProvider : public wxArtProvider
{
protected:
    virtual wxBitmap CreateBitmap(const wxArtID& id,
                                  const wxArtClient& client,
                                  const wxSize& size);
};
// 新的CreateBitmap函数
wxBitmap MyArtProvider::CreateBitmap(const wxArtID& id,
                                     const wxArtClient& client,
                                     const wxSize& size)
{
    if (id == wxART_HELP_SIDE_PANEL)
        return wxBitmap(helpsidepanel_xpm);
    if (id == wxART_HELP_SETTINGS)
        return wxBitmap(helpoptions_xpm);
    if (id == wxART_HELP_BOOK)
        return wxBitmap(helpbook_xpm);
    if (id == wxART_HELP_FOLDER)
        return wxBitmap(helpbook_xpm);
    if (id == wxART_HELP_PAGE)
        return wxBitmap(helppage_xpm);
    if (id == wxART_GO_BACK)
        return wxBitmap(helpback_xpm);
    if (id == wxART_GO_FORWARD)
        return wxBitmap(helpforward_xpm);
    if (id == wxART_GO_UP)
        return wxBitmap(helpup_xpm);
    if (id == wxART_GO_DOWN)
        return wxBitmap(helpdown_xpm);
    if (id == wxART_GO_TO_PARENT)
        return wxBitmap(helpuplevel_xpm);
    if (id == wxART_FRAME_ICON)
        return wxBitmap(helpicon_xpm);
    if (id == wxART_HELP)
        return wxBitmap(helpicon_xpm);

    // Any wxWidgets icons not implemented here
    // will be provided by the default art provider.
    return wxNullBitmap;
}
// 你的初始化函数
bool MyApp::OnInit()
{
    ...
    wxArtProvider::PushProvider(new MyArtProvider);
    ...
    return true;
}
```

第十章小结

在这一章里,我们学习了怎样使用wxWidgets中的图片相关的类wxBitmap, wxIcon, wxCursor和wxImage,还学习了怎样使用wxImageList和wxIconBundle,以及怎样定义wxWidgets默认使用的小图片.更多相关的例子请参考wxWidgets自带的samples/image, samples/listctrl和samples/dragimag目录中的例子.

在下一章里,我们将介绍一下怎样使用剪贴板来传输数据以及怎样实现拖放编程.

第十一章剪贴板和拖放操作

大多数应用程序通常都会从剪贴板通过拷贝,剪切和粘贴的动作来实现数据的传输.这是你的应用程序和别的应用程序进行交互的一种最基本的方式,大多数老道的应用程序还会允许使用者在一个程序内部的窗口之间或者是不同的应用程序之间进行拖放操作.比如把一个文件从资源管理器拖放到应用程序窗口,以便这个窗口直接填充这个文件相关的数据.或者是将这个文件名添加到某个列表中,或者执行别的什么行为.这种操作比起通过菜单打开一个对话框来选择文件的方式要快捷的多,你的用户将会很喜欢这样的操作方式.

剪贴板和拖放操作在wxWidgets中共享了一些类,这主要是因为他们的主要任务都是实现数据传输,因此本章把这两个操作放在一起操作.我们将会看到怎样使用wxWidgets提供的标准数据传输控件,同时也会学习到怎样实现自己的数据传输控件.

11.1 数据对象

wxDataObject类是剪贴板操作和拖放操作的核心,这个类的实例代表了拖放操作中鼠标拖拽的事物,以及剪贴板操作中拷贝和粘贴的事物. wxDataObject是一块聪明的数据,因为它知道哪种格式它可以支持(通过GetFormatCount和GetAllFormats),也知道怎样来支持它们(通过GetdataHere).如果实现了对应的SetData函数,它也可以从应用程序的外部接受不同格式的数据.我们将在本章的后边对此进行介绍.

标准的数据格式,比如wxDF_TEXT是用整数来区分的,而定制的数据格式则是通过字符串来区分的.wxDataFormat类支持使用这两种参数的构造函数.下表列出了标准的数据格式.

wxDF_INVALID	无效格式,用于缺省的**wxDataFormat**的构造函数.
wxDF_TEXT	文本数据格式,对应的数据类型为wxTexTDataObject.
wxDF_BITMAP	图片数据格式,对应的数据类型为wxBitmapDataObject.
wxDF_METAFILE	元文件数据格式,对应的数据类型为wxMetafileDataObject(仅支持Windows)
wxDF_FILENAME	文件名列表数据格式,对应的数据类型为wxFileDataObject.

你也可以创建定制的数据格式,在这种情况下,你需要给wxDataFormat构造函数传递一个定制的字符串,来标识你的定制的数据类型,这个数据类型将在首次使用的时候被登记.

剪贴板操作和拖放操作都需要处理数据源(数据提供者)和数据目标(数据接收者).它们可以位于同一个应用程序内,甚至是位于同一个窗口内,比如,你在同一个窗口内把一段文本从一个位置拖到另一个位置,我们来分别描述一下这两个角色.

数据源的职责

数据源负责创建要包含要传输的数据的数据对象,在创建数据对象以后,数据源还负责通过SetData函数将其传递给剪贴板,或者在拖放操作开始时,通过DoDragDrop函数将其传递给一个wxDropSource对象.

在这种情况下,剪贴板操作和拖放操作的最大的不同在于剪贴板传输的数据必须使用new函数,在堆上创建,而且只能被剪贴板在其不被需要的时候释放,事实上,我们根本不知道它是在什么时候被释放的,我们甚至连原始的数据是什么时候被放到剪贴板上去的也不知道.而另一方面,用于拖放操作的数据对象只需要在DoDragDrop执行期间存在,执行完以后就可以被安全地释放了,因此,这种数据对象,即可以在堆上创建,也可以在栈上创建(意思就是一个局部变量).

另一个细微的差别在于:对于剪贴板操作应用程序通常很清楚它正在进行的操作的整个过程.当进行了剪切操作的时候,数据被首先拷贝到剪切板,然后从当前操作的对象中移除.这通常是由于用户对菜单项的选择来触发的.但是对于拖放操作来说,应用程序只有在DoDragDrop函数执行以后,才能了解这些信息.

数据目标的职责

要从剪贴板接收数据(意味着一个粘贴操作),你应该首先创建一个支持你想要获取的数据格式的wxDataObject的派生类,以便将其传递给wxClipboard::GetData函数.如果这个函数返回失败,表明剪贴板上没有你想要的类型的数据.如果返回成功,则表明剪贴板上的数据已经被成功地传输到你创建的wxDataObject的派生类中.

对于拖放操作,当一个数据对象被放置的时候,wxDropTarget::OnData虚函数将会被调用.如果数据类型合适,应用程序可以调用wxDropTarget::OnData函数来获取相应的数据.

11.2 使用剪贴板

要使用剪贴板,你主要是在调用全局指针wxTheClipboard的成员函数. 在进行拷贝或者粘贴的动作之前,你必须先通过wxClipboard::Open获得剪贴板的控制权,如果这个函数返回成功,你将已经获得了剪贴板的控制权,可以调用wxClipboard::SetData来将数据拷贝到剪贴板上,或者调用wxClipboard::GetData函数从剪贴板上获取数据.最后,你需要调用wxClipboard::Close函数来释放剪贴板的控制权.一旦你不使用剪贴板了,就应该尽快释放掉剪贴板的控制权.

wxClipboardLocker类可以在其构造函数中获得剪贴板的控制权(如果可以的话),并且在其析构函数中释放剪贴板的控制权,因此,你可以使用下面这样的代码:

```
wxClipboardLocker locker;
if (!locker)
{
   ... 报告错误然后返回 ...
}
... 使用剪贴板 ...

下边的代码演示了怎样将文本拷贝到剪贴板以及怎样从剪贴板读取文本数据:

// 拷贝一些文本到剪贴板
if (wxTheClipboard->Open())
{
    // 数据对象将被剪贴板释放,
    // 因此不在要你的应用程序中释放它们.
    wxTheClipboard->SetData(new wxTextDataObject(wxT("Some text")));
    wxTheClipboard->Close();
}

// 从剪贴板获取一些文本
if (wxTheClipboard->Open())
{
    if (wxTheClipboard->IsSupported(wxDF_TEXT))
    {
        wxTextDataObject data;
        wxTheClipboard->GetData(data);
        wxMessageBox(data.GetText());
    }
    wxTheClipboard->Close();
}

下边是一个操作图片的例子:

// 将一副图片拷贝到剪贴板
wxImage image(wxT("splash.png"), wxBITMAP_TYPE_PNG);
wxBitmap bitmap(image.ConvertToBitmap());
if (wxTheClipboard->Open())
{
    // 数据对象将被剪贴板释放,
    // 因此不在要你的应用程序中释放它们.
    wxTheClipboard->SetData(new wxBitmapDataObject(bitmap));
    wxTheClipboard->Close();
}

// 从剪贴板读取一幅图片
if (wxTheClipboard->Open())
{
    if (wxTheClipboard->IsSupported(wxDF_BITMAP))
    {
        wxBitmapDataObject data;
        wxTheClipboard->GetData( data );
        bitmap = data.GetBitmap();
    }
    wxTheClipboard->Close();
}
```

如果你使用了剪贴板操作你可能需要即时更新你的用户界面,比如在剪贴板拥有或者失去相关数据的时候,允许或者禁用相关的菜单项,工具条上的按钮以及一些普通的按钮.这个工作是通过wxWidgets的界面更新机制来完成的.在合适的时候 wxWidgets将会给你的应用程序发送wxUpdateUIEvent事件,详情请参考第九章"创建自己自定义的对话框".这个事件允许你在系统空闲的时候根据剪贴板的数据来更新你的用户界面.

某些控件,比如wxTextCtrl已经实现了用户界面的自动更新.如果你的菜单项或者工具条使用了标准的标识符wxID_CUT, wxID_COPY, 和wxID_PASTE,并且指定了命令事件将首先被活动的控件处理,那么对应的控件将会自动按照用户的预期来进行界面更新.参考第二十章"优化你的应用程序"来学习怎样通过重载wxFrame::ProcessEvent函数将命令事件传递到当前激活的控件.

11.3 实现拖放操作

你可以在你的应用程序中实现拖放源,拖放目标或者两者同时实现.

实现拖放源

要实现一个拖放源,也就是说要提供用户用于拖放操作的数据,你需要使用一个wxDropSource类的实例.要注意下面描述的事情都是在你的应用程序已经认定一个拖放操作已经开始以后发生的.决定拖放是否开始的逻辑,是完全需要由应用程序自己决定的,一些控件会通过产生一个拖放开始事件来通知应用程序, 在这种情况下,你不需要自己关心这一部分的逻辑(对这一部分的逻辑的关心,可能会让你的鼠标事件处理代码变得混乱).在本章中,也会提供一个何时wxWidgets通知你拖放操作开始的大概描述.

一个拖放源需要采取的动作包括下面几步:

1 准备工作

首先,必须先创建和初始化一个将被拖动的数据对象,如下所示:

```
wxTextDataObject myData(wxT("This text will be dragged."));
```

2 开始拖动要开始拖动操作,最典型的方式是响应一个鼠标单击事件,创建一个wxDropSource对象,然后调用它的wxDropSource::DoDragDrop函数,如下所示:

```
wxDropSource dragSource(this);
dragSource.SetData(myData);
wxDragResult result = dragSource.DoDragDrop(wxDrag_AllowMove);
```

下表列出的标记,可以作为DoDragDrop函数的参数:

wxDrag_CopyOnly	只允许进行拷贝操作.
wxDrag_AllowMove	允许进行移动操作.
wxDrag_DefaultMove	默认操作是移动数据.

当创建wxDropSource对象的时候,你还可以指定发起拖动操作的窗口,并且可以选择拖动使用的光标,可选的范围包括拷贝,移动以及不能释放等.这些光标在GTK+上是图标,而在别的平台上是光标,因此你需要使用wxDROP_ICON来屏蔽这种区别,正如我们很快就将看到的拖动文本的例子中演示的那样.

3 拖动

对DoDragDrop函数的调用将会阻止应用程序进行其他处理,直到用户释放鼠标按钮(除非你重载了GiveFeedback函数以便进行其他的特殊操作)当鼠标在应用程序的一个窗口上移动时,如果这个窗口可以识别这个拖动操作协议,对应的wxDropTarget函数就会被调用,参考接下来的小节"实现一个拖放目的"

4 处理拖放结果

DoDragDrop函数返回一个拖放操作的结果,这个返回值的类型为wxDragResult,它的枚举值如下表所示:

wxDragError	在拖动操作的执行过程中出现了错误.
wxDragNone	数据源不被拖动目的接受.
wxDragCopy	数据已经被成功拷贝.
wxDragMove	数据已经被成功移动(仅适用于Windows).
wxDragLink	以完全一个链接操作.
wxDragCancel	用户已经取消了拖放操作.

你的应用程序可以针对不同的返回值进行自己的操作,如果返回值是wxDragMove,通常你需要删除绑定在数据源中的数据,然后更新屏幕显示.而如果返回值是wxDragNone,则表示拖动操作已经被取消了.下面举例说明:

```
switch (result)
{
    case wxDragCopy: /* 数据被拷贝或者被链接:
                          无需特别操作 */
    case wxDragLink:
        break;
    case wxDragMove: /* 数据被移动,删除原始数据 */
        DeleteMyDraggedData();
        break;
    default:         /* 操作被取消或者数据不被接受
                         或者发生了错误:
                         不做任何操作 */
        break;
}
```

下面的例子演示了怎样实现一个文本数据拖放源.DnDWindow包含一个m_strText成员变量,当鼠标左键按下的时候,针对 m_strText的拖放操作开始,拖放操作的结果通过一个消息框显示.另外,拖放操作将会在鼠标已经拖动了一小段距离后才会开始,因此单击鼠标动作并不会导致一个拖放操作.

```
void DnDWindow::OnLeftDown(wxMouseEvent& event )
{
    if ( !m_strText.IsEmpty() )
    {
        // 开始拖动操作
        wxTextDataObject textData(m_strText);
        wxDropSource source(textData, this,
                            wxDROP_ICON(dnd_copy),
                            wxDROP_ICON(dnd_move),
                            wxDROP_ICON(dnd_none));
        int flags = 0;
        if ( m_moveByDefault )
            flags |= wxDrag_DefaultMove;
        else if ( m_moveAllow )
            flags |= wxDrag_AllowMove;
        wxDragResult result = source.DoDragDrop(flags);
        const wxChar *pc;
        switch ( result )
        {
            case wxDragError:   pc = wxT("Error!");    break;
            case wxDragNone:    pc = wxT("Nothing");   break;
            case wxDragCopy:    pc = wxT("Copied");    break;
            case wxDragMove:    pc = wxT("Moved");     break;
            case wxDragCancel:  pc = wxT("Cancelled"); break;
            default:            pc = wxT("Huh?");      break;
        }
        wxMessageBox(wxString(wxT("Drag result: ")) + pc);
    }
}
```

实现一个拖放目的

要实现一个拖放目的,也就是说要接收用户拖动的数据,你需要使用wxWindow::SetDropTarget函数,将某个窗口和一个 wxDropTarget绑定在一起,你需要实现一个wxDropTarget的派生类,并且重载它的所有纯虚函数.另外还需要重载OnDragOver 函数,以便返回一个wxDragResult类型的返回码,以说明当鼠标指针移过这个窗口的时候,光标应该怎样显示,并且重载OnData函数来实现放置操作.你还可以通过继承wxTexTDropTarget或者wxFileDropTarget,或者重载它们的OnDropText或者 OnDropFiles函数来实现一个拖放目的.

下面的步骤将发生在拖放操作过程当中的拖放目的对象上.

1 初始化

wxWindow::SetDropTarget函数在窗口创建期间被调用,以便将其和一个拖放目的对象绑定.在窗口创建或者应用程序的其他某个部分,通过函数wxDropTarget::SetDataObject,拖放目的对象和某一种数据类型绑定,这种数据类型将用来作为拖放源和播放目的进行协商的依据.

2 拖动

当鼠标在拖放目的上以拖动的方式移动时,wxDropTarget::OnEnter,wxDropTarget::OnDragOver和 wxDropTarget::OnLeave函数将在适当的时候被调用,它们都将返回一个对应的wxDragResult值.以便拖放操作可以对其进行合适的用户界面反馈.

3 放置

当用户释放鼠标按钮的时候,wxWidgets通过调用函数wxDataObject::GetAllFormats询问窗口绑定的 wxDropTarget对象是否接受正在拖动的数据.如果数据类型是可接受的,那么wxDropTarget::OnData将被调用.拖放对象绑定的 wxDataObject对象将进行对应的数据填充动作.wxDropTarget::OnData函数将返回一个wxDragResult类型的值,这个值将作为wxDropSource::DoDragDrop函数的返回值.

使用标准的拖放目的对象

wxWidgets提供了了标准的wxDropTarget的派生类,因此你不必在任何时候都需要实现自己的拖放对象.你只需要实现重载这些类的一个虚函数,以便在拖放的时候得到提示.

wxTextdropTarget对象可以接收被拖动的文本数据,你只需要重载OnDropText函数以便告诉wxWidgets当有文本数据被放置的时候做什么事情就可以了.下面的例子演示了当有文本数据被放置的时候,怎样将其添加列表框内.

```
// 一个拖放目的用来将文本填加到列表框
class DnDText : public wxTextDropTarget
{
public:
    DnDText(wxListBox *owner) { m_owner = owner; }
    virtual bool OnDropText(wxCoord x, wxCoord y,
                            const wxString& text)
    {
        m_owner->Append(text);
        return true;
    }
private:
    wxListBox *m_owner;
};
// 设置拖放目的对象
wxListBox* listBox = new wxListBox(parent, wxID_ANY);
listBox->SetDropTarget(new DnDText(listBox));
```

下面的例子展示了怎样使用wxFileDropTarget,这个对象可接收从资源管理器里拖动的文件对象,并且报告拖动文件的数目以及它们的名称.

```
// 一个拖放目的类用来将拖动的文件名添加到列表框
class DnDFile : public wxFileDropTarget
{
public:
    DnDFile(wxListBox *owner) { m_owner = owner; }
    virtual bool OnDropFiles(wxCoord x, wxCoord y,
                             const wxArrayString& filenames)
    {
        size_t nFiles = filenames.GetCount();
        wxString str;
        str.Printf( wxT("%d files dropped"), (int) nFiles);
        m_owner->Append(str);
        for ( size_t n = 0; n &lt; nFiles; n++ ) {
            m_owner->Append(filenames[n]);
        }
        return true;
    }
private:
    wxListBox *m_owner;
};
// 设置拖放目的类
wxListBox* listBox = new wxListBox(parent, wxID_ANY);
listBox->SetDropTarget(new DnDFile(listBox));
```

创建一个自定义的拖放目的

现在我们来创建一个自定义的拖放目的,它可以接受URLs(网址).这一次我们需要重载OnData和 OnDragOver函数,我们还将实现一个可以被重载的虚函数OnDropURL.

```
// 一个自定义的拖放目的对象,可以拖放URL对象
class URLDropTarget : public wxDropTarget
{
public:
    URLDropTarget() { SetDataObject(new wxURLDataObject); }
    void OnDropURL(wxCoord x, wxCoord y, const wxString& text)
    {
        // 当然, 一个真正的应用程序在这里应该做些更有意义的事情
        wxMessageBox(text, wxT("URLDropTarget: got URL"),
                     wxICON_INFORMATION | wxOK);
    }
    // URLs不能被移动,只能被拷贝
    virtual wxDragResult OnDragOver(wxCoord x, wxCoord y,
                                    wxDragResult def)
        {
            return wxDragLink;
        }
    // 这个函数调用了OnDropURL函数,以便它的派生类可以更方便的使用
    virtual wxDragResult OnData(wxCoord x, wxCoord y,
                                wxDragResult def)
    {
        if ( !GetData() )
            return wxDragNone;
        OnDropURL(x, y, ((wxURLDataObject *)m_dataObject)->GetURL());
        return def;
    }
};
// 设置拖放目的对象
wxListBox* listBox = new wxListBox(parent, wxID_ANY);
listBox->SetDropTarget(new URLDropTarget);
```

更多关于wxDataObject的知识

正如我们已经看到的那样,wxDataObject用来表示所有可以被拖放.以及可以被剪贴板操作的数据.wxDataObject最重要的特性之一,是在于它是一个"聪明"的数据块,和普通的包含一段内存缓冲,或者一些文件的哑数据不同.所谓"聪明"指的是数据对像自己可以知道它内部的数据可以支持什么数据格式,以及怎样将它的内部数据表现为那种数据格式.

所谓支持的数据格式,意思是说,这种格式可以从一个数据对象的内部数据或者将被设置的内部数据产生.通常情况下,一个数据对象可以支持多种数据格式作为输入或者输出.因此,一个数据对象可以支持从一种格式创建内部数据,并将其转换为另外一种数据格式,反之亦然.

当你需要使用某种数据对象的时候,有下面几种可选方案:

1. 使用一种内建的数据对象.当你只需要支持文本数据,图片数据,或者是文件列表数据的时候,你可以使用wxTextdataObject,wxBitmapDataObject,或wxFileDataObject.
2. 使用wxDataObjectSimple.从wxDataObjectSimple产生一个派生类,是实现自定义数据格式的最简便的方法,不过,这种派生类只能支持一种数据格式,因此,你将不能够使用它和别的应用程序进行交互.不过,你可以用它在你的应用程序以及你应用程序的不同拷贝之间进行数据传输.
3. 使用wxCustomDataObject的派生类(它是wxDataObjectSimple的一个子类)来实现自定义数据对象.
4. 使用wxDataObjectComposite.这是一个简单而强大的解决方案,它允许你支持任意多种数据格式(如果你和前面的方法结合使用的话,可以实现同时支持标准数据和自定义数据).
5. 直接使用wxDataObject.这种方案可以实现最大的灵活度和效率,但也是最难的一种实现方案.

在拖放操作和剪贴板操作中,使用多重数据格式最简单的方法是使用wxDataObjectComposite对象,但是,这种使用方法的效率是很低的,因为每一个wxDataObjectSimple都使用自己定义的格式来保存所有的数据.试想一下,你将从剪贴板上以你自己的格式粘贴超过两百页的文本,其中包含Word,RTF,HTML,Unicode,和普通文本,虽然现在计算机的能力已经足可以应付这样的任务,但是从性能方面考虑, 你最好还是直接从wxDataObject实现一个派生类,用它来定义你的数据格式,然后在程序中指定这种类型的数据.

剪贴板操作和拖放操作,潜在的数据传输机制将在某个应用程序真正请求数据的时候,才会进行数据拷贝.因此,尽管用户可能认为在自己点击了应用程序的拷贝按钮以后数据就已存在于剪贴板了,但实际上这时候仅仅是告诉剪贴板有数据存在了.

实现wxDataObject的派生类

我们来看一下实现一个新的wxDataObject的派生类要用到哪些东西.至于怎样实现的过程,我们前面已经介绍过了,它是非常简单的.因此,我们在这里不多说了.

每一个wxDataObject的派生类,都必须重载和实现它的纯虚成员函数,那些只能用来输出数据或者保存数据(意味着只能进行单向操作)的数据对象的GetFormatCount函数在其不支持的方向上应该总是返回零.

GetAllFormats函数的参考为一个wxDataFormat类型的列表,以及一个方向(获取或设置).它将所有自己在这个方向上支持的数据格式填入这个列表.GetFormatCount函数则用来检测列表中元素的个数.

GetdataHere函数的参数是一个wxDataFormat参数,以及一个void*缓冲区.如果操作成功,返回TRue,否则返回false.这个函数必须将数据以给定的格式填入这个缓冲区,数据可以是任意的二进制数据,或者是文本数据,只要SetData函数可以识别就行了.

GetdataSize函数则返回在某种给定的数据格式下数据的大小.

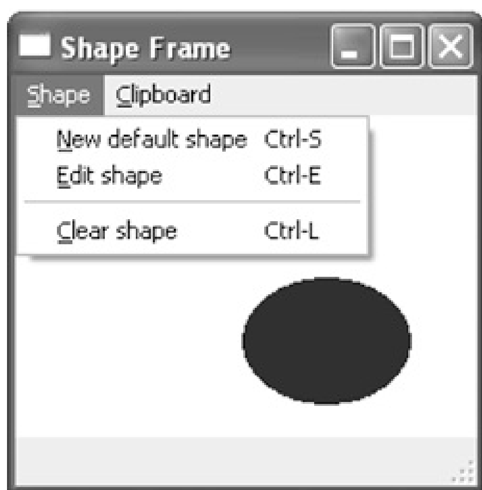
GetFormatCount函数返回用于转换或者设置的当前支持数据类型的个数.

GetPreferredFormat函数则返回某个指定方向上优选的数据类型.

SetData函数的参考包括一个数据类型,一个整数格式的缓冲区大小,以及一个void*类型的缓冲区指针.你可以在适当的时候(比如将其拷贝到自己的内部结构的时候)对其进行适当的解释.这个函数在成功的时候返回TRue,失败的时候返回false.

wxWidgets的拖放操作例子

我们通过使用位于samples/dnd目录的wxWidgets的拖放操作的例子,来演示一下怎样制作一个自定义的拥有自定义数据类型的数据对象.这个例子演示了一个简单的绘图软件,可以用来绘制矩形,三角形,或者椭圆形,并且允许你对其进行编辑,拖放到一个新的位置,拷贝到剪贴板以及从新粘贴到应用程序等操作.你可以通过选择文件菜单中的新建命令来创建一个应用程序的主窗口.这个窗口的外观如下图所示:



这些图形是用继承字DnDShape的类来建模的,数据对象被称为DnDShapeDataObject.在学习怎样实现这个数据对象之前,我们先看一下应用程序是怎样使用它们的.

当一个剪贴板拷贝操作被请求的时候,一个DnDShapeDataObject对象将被增加到剪贴板,这个对象包含当前正在操作的图形的拷贝,如果剪贴板上已经有了一个对象,那么旧的对象将被释放.代码如下所示:

```
void DnDShapeFrame::OnCopyShape(wxCommandEvent& event)
{
    if ( m_shape )
    {
        wxClipboardLocker clipLocker;
        if ( !clipLocker )
        {
            wxLogError(wxT("Can't open the clipboard"));
            return;
        }
        wxTheClipboard->AddData(new DnDShapeDataObject(m_shape));
    }
}
```

剪贴板的粘贴操作也很容易理解,调用wxClipboard::GetData函数来获取位于剪贴板的图形数据对象,然后从其中获取图形数据.前面我们已经介绍过怎样在剪贴板拥有对应数据的时候允许paste菜单.shapeFormatId是一个全局变量,其中包含了自定义的数据格式名称:wxShape.

```
void DnDShapeFrame::OnPasteShape(wxCommandEvent& event)
{
    wxClipboardLocker clipLocker;
    if ( !clipLocker )
    {
        wxLogError(wxT("Can't open the clipboard"));
        return;
    }
    DnDShapeDataObject shapeDataObject(NULL);
    if ( wxTheClipboard->GetData(shapeDataObject) )
    {
        SetShape(shapeDataObject.GetShape());
    }
    else
    {
        wxLogStatus(wxT("No shape on the clipboard"));
    }
}
void DnDShapeFrame::OnUpdateUIPaste(wxUpdateUIEvent& event)
{
    event.Enable( wxTheClipboard->
                  IsSupported(wxDataFormat(shapeFormatId)) );
}
```

为了实现拖放操作,还需要一个拖放目标对象,以便在图片数据被放置的时候通知应用程序.DnDShapeDropTarget类包含一个 DnDShapeDataObject数据对象,用来扮演这个角色,当它的OnData函数被调用的时候,则表明正在放置一个图形数据对象,下面的代码演示了DnDShapeDropTarget的声明及实现:

```
class DnDShapeDropTarget : public wxDropTarget
{
public:
    DnDShapeDropTarget(DnDShapeFrame *frame)
        : wxDropTarget(new DnDShapeDataObject)
    {
        m_frame = frame;
    }
    // 重载基类的(纯)虚函数
    virtual wxDragResult OnEnter(wxCoord x, wxCoord y, wxDragResult def)
    {
        m_frame->SetStatusText(_T("Mouse entered the frame"));
        return OnDragOver(x, y, def);
    }
    virtual void OnLeave()
    {
        m_frame->SetStatusText(_T("Mouse left the frame"));
    }
    virtual wxDragResult OnData(wxCoord x, wxCoord y, wxDragResult def)
    {
        if ( !GetData() )
        {
            wxLogError(wxT("Failed to get drag and drop data"));
            return wxDragNone;
        }
        // 通知主窗口正在进行放置
        m_frame->OnDrop(x, y,
                ((DnDShapeDataObject *)GetDataObject())->GetShape());
        return def;
    }
private:
    DnDShapeFrame *m_frame;
};
```

在应用程序初始化函数里,主窗口被创建的时候设置这个拖放目标:

```
DnDShapeFrame::DnDShapeFrame(wxFrame *parent)
             : wxFrame(parent, wxID_ANY, _T("Shape Frame"))
{
    ...
    SetDropTarget(new DnDShapeDropTarget(this));
    ...
}
```

当鼠标左键单击的时候,拖放操作开始,其处理函数将创建一个wxDropSource对象,并且给DoDragDrop 函数传递一个DnDShapeDataObject对象,以便初始化拖放操作.DndShapeFrame::OnDrag函数如下所示:

```
void DnDShapeFrame::OnDrag(wxMouseEvent& event)
{
    if ( !m_shape )
    {
        event.Skip();
        return;
    }
    // 开始拖放操作
    DnDShapeDataObject shapeData(m_shape);
    wxDropSource source(shapeData, this);
    const wxChar *pc = NULL;
    switch ( source.DoDragDrop(true) )
    {
        default:
        case wxDragError:
            wxLogError(wxT("An error occured during drag and drop"));
            break;
        case wxDragNone:
            SetStatusText(_T("Nothing happened"));
            break;
        case wxDragCopy:
            pc = _T("copied");
            break;
        case wxDragMove:
            pc = _T("moved");
            if ( ms_lastDropTarget != this )
            {
                // 如果这个图形被放置在自己的窗口上
                // 不要删除它
                SetShape(NULL);
            }
            break;
        case wxDragCancel:
            SetStatusText(_T("Drag and drop operation cancelled"));
            break;
    }
    if ( pc )
    {
        SetStatusText(wxString(_T("Shape successfully ")) + pc);
    }
    //在其他情况下,状态文本已经被设置了
}
```

当用户释放鼠标以表明正在执行放置操作的时候,wxWidgets调用DnDShapeDropTarget::OnData函数,这个函数将以一个新的DndShape对象来调用DndShapeFrame::OnDrop函数,以便给DndShape对象设置一个新的位置.这样,拖放操作就完成了.

```
void DnDShapeFrame::OnDrop(wxCoord x, wxCoord y, DnDShape *shape)
{
    ms_lastDropTarget = this;
    wxPoint pt(x, y);
    wxString s;
    s.Printf(wxT("Shape dropped at (%d, %d)"), pt.x, pt.y);
    SetStatusText(s);
    shape->Move(pt);
    SetShape(shape);
}
```

现在,唯一剩下的事情,就是实现自定义的wxDataObject对象了.为了说明得更清楚,我们将对整个实现进行逐项说明.首先,我们来看一下自定义数据类型标识符声明,以及DndShapeDataObject类的声明,它的构造函数和析构函数,和它的数据成员.

数据类型标识符是shapeFormatId,它是一个全局变量,在整个例子中都有使用.构造函数通过GeTDataHere函数获得当前图形(如果有的话)的一个拷贝作为参数.这个拷贝也可以通过DndShape::Clone函数产生.DnDShapeDataObject的析构函数将会释放这个拷贝.

DndShapeDataObject可以提供位图和(在支持的平台上)源文件来表示它的内部数据.因此,它还拥有 wxBitmapDataObject和wxMetaFileDataObject两个类型的数据成员(以及一个标记用来指示当前正在使用哪种类型)来缓存内部数据以便在需要的时候提供这种格式.

```
// 自定义的数据格式标识符
static const wxChar *shapeFormatId = wxT("wxShape");

class DnDShapeDataObject : public wxDataObject
{
public:
    // 构造函数没有直接拷贝指针
    // 这样在原来的图形对象被释放以后,这里的图形对象是有效的
    DnDShapeDataObject(DnDShape *shape = (DnDShape *)NULL)
    {
        if ( shape )
        {
            // 我们需要拷贝真正的图形对象,而不是只拷贝指针
            // 这是因为图形对象有可能在任何时候被删除,在这种情况下
            // 剪贴板上的数据仍然应该是有效的
            // 因此我们使用下边的方法来实现图形拷贝
            void *buf = malloc(shape->DnDShape::GetDataSize());
            shape->GetDataHere(buf);
            m_shape = DnDShape::New(buf);
            free(buf);
        }
        else
        {
            // 不需要拷贝任何东西
            m_shape = NULL;
        }
        // 这个字符串应该用来唯一标识我们的数据格式类型
        // 除此以外,它可以是任意的字符串
        m_formatShape.SetId(shapeFormatId);
        // 我们直到需要的(也就是数据被第一次请求)时候才产生图片或者元文件数据
        m_hasBitmap = false;
        m_hasMetaFile = false;
    }
    virtual ~DnDShapeDataObject() { delete m_shape; }
    // 在这个函数被调用以后,图形数据归调用者所有
    // 调用者将负责释放相关内存
    DnDShape *GetShape()
    {
        DnDShape *shape = m_shape;
        m_shape = (DnDShape *)NULL;
        m_hasBitmap = false;
        m_hasMetaFile = false;
        return shape;
    }
    // 其他成员函数省略
    ...
    // 数据成员
private:
    wxDataFormat         m_formatShape; // 我们的自定义格式
    wxBitmapDataObject   m_dobjBitmap;  // 用来响应位图格式请求
    bool                 m_hasBitmap;   // 如果m_dobjBitmap有效为真
    wxMetaFileDataObject m_dobjMetaFile;// 用来响应元数据格式请求
    bool                 m_hasMetaFile;// 如果m_dobjMetaFile有效为真
    DnDShape             *m_shape;       // 原始数据
};
```

接下来我们来看一下那些用于回答和我们内部存储的数据相关的问题的函数.GetPreferredFormat只简单的返回 m_formatShape数据绑定的本地的数据格式,它是在我们的构造函数中使用wxShape类型初始化的.GetFormatCount函数用来检测某种特定的格式是否可以被用来获取或者设置数据.在获取数据的时候,只有位图和元文件格式是可以被处理的.GetDataSize函数依据请求的数据格式的不同返回合适的数据大小,如果必要的话,为了得到这个大小,你可以在这个时候创建位图成员或者元文件成员.

```
virtual wxDataFormat GetPreferredFormat(Direction dir) const
{
    return m_formatShape;
}
virtual size_t GetFormatCount(Direction dir) const
{
    // 我们自定义的数据格式类型即可以支持 GetData()
    // 也可以支持 SetData()
    size_t nFormats = 1;
    if ( dir == Get )
    {
        // 但是,位图格式只支持输出
        nFormats += m_dobjBitmap.GetFormatCount(dir);
        nFormats += m_dobjMetaFile.GetFormatCount(dir);
    }
    return nFormats;
}
virtual void GetAllFormats(wxDataFormat *formats, Direction dir) const
{
    formats[0] = m_formatShape;
    if ( dir == Get )
    {
        // 在获取方向上我们增加位图和元文件两种格式的支持
        //在Windows平台上
        m_dobjBitmap.GetAllFormats(&formats[1], dir);
        // 不要认为m_dobjBitmap只有一种格式
        m_dobjMetaFile.GetAllFormats(&formats[1 +
                m_dobjBitmap.GetFormatCount(dir)], dir);
    }
}
virtual size_t GetDataSize(const wxDataFormat& format) const
{
    if ( format == m_formatShape )
    {
        return m_shape->GetDataSize();
    }
    else if ( m_dobjMetaFile.IsSupported(format) )
    {
        if ( !m_hasMetaFile )
            CreateMetaFile();
        return m_dobjMetaFile.GetDataSize(format);
    }
    else
    {
        wxASSERT_MSG( m_dobjBitmap.IsSupported(format),
                      wxT("unexpected format") );
        if ( !m_hasBitmap )
            CreateBitmap();
        return m_dobjBitmap.GetDataSize();
    }
}
```

GetdataHere函数按照请求的数据格式类型将数据拷贝到void*类型的缓冲区:

```
virtual bool GetDataHere(const wxDataFormat& format, void *pBuf) const
{
    if ( format == m_formatShape )
    {
        // 使用ShapeDump结构将其转换为void*流
        m_shape->GetDataHere(pBuf);
        return true;
    }
    else if ( m_dobjMetaFile.IsSupported(format) )
    {
        if ( !m_hasMetaFile )
            CreateMetaFile();
        return m_dobjMetaFile.GetDataHere(format, pBuf);
    }
    else
    {
        wxASSERT_MSG( m_dobjBitmap.IsSupported(format),
                      wxT("unexpected format") );
        if ( !m_hasBitmap )
            CreateBitmap();
        return m_dobjBitmap.GetDataHere(pBuf);
    }
}
```

SetData函数只需要处理本地格式,因此,它需要做的所有事情就是使用DndShape::New函数来制作一个参数图形的拷贝:

```
virtual bool SetData(const wxDataFormat& format,
                       size_t len, const void *buf)
{
    wxCHECK_MSG( format == m_formatShape, false,
                 wxT( "unsupported format") );
    delete m_shape;
    m_shape = DnDShape::New(buf);
    // the shape has changed
    m_hasBitmap = false;
    m_hasMetaFile = false;
    return true;
}
```

实现DndShape和void*类型的互相转换的方法是非常直接的.它使用了一个ShapeDump的结构来保存图形的详细信息.下面是其实现方法:

```
// 静态函数用来从一个void*缓冲区中创建一个图形
DnDShape *DnDShape::New(const void *buf)
{
    const ShapeDump& dump = *(const ShapeDump *)buf;
    switch ( dump.k )
    {
        case Triangle:
            return new DnDTriangularShape(
                             wxPoint(dump.x, dump.y),
                             wxSize(dump.w, dump.h),
                             wxColour(dump.r, dump.g, dump.b));
        case Rectangle:
            return new DnDRectangularShape(
                             wxPoint(dump.x, dump.y),
                             wxSize(dump.w, dump.h),
                             wxColour(dump.r, dump.g, dump.b));
        case Ellipse:
            return new DnDEllipticShape(
                             wxPoint(dump.x, dump.y),
                             wxSize(dump.w, dump.h),
                             wxColour(dump.r, dump.g, dump.b));
        default:
            wxFAIL_MSG(wxT("invalid shape!"));
            return NULL;
    }
}
// 返回内部数据大小
size_t DndShape::GetDataSize() const
{
    return sizeof(ShapeDump);
}
// 将自己填入一个void*缓冲区
void DndShape::GetDataHere(void *buf) const
{
    ShapeDump& dump = *(ShapeDump *)buf;
    dump.x = m_pos.x;
    dump.y = m_pos.y;
    dump.w = m_size.x;
    dump.h = m_size.y;
    dump.r = m_col.Red();
    dump.g = m_col.Green();
    dump.b = m_col.Blue();
    dump.k = GetKind();
}
```

最后,我们回到DnDShapeDataObject数据对象,下边的这些函数用来在需要的时候将内部数据转换为位图或者元数据:

```
void DnDShapeDataObject::CreateMetaFile() const
{
    wxPoint pos = m_shape->GetPosition();
    wxSize size = m_shape->GetSize();
    wxMetaFileDC dcMF(wxEmptyString, pos.x + size.x, pos.y + size.y);
    m_shape->Draw(dcMF);
    wxMetafile *mf = dcMF.Close();
    DnDShapeDataObject *self = (DnDShapeDataObject *)this;
    self->m_dobjMetaFile.SetMetafile(*mf);
    self->m_hasMetaFile = true;
    delete mf;
}
void DnDShapeDataObject::CreateBitmap() const
{
    wxPoint pos = m_shape->GetPosition();
    wxSize size = m_shape->GetSize();
    int x = pos.x + size.x,
        y = pos.y + size.y;
    wxBitmap bitmap(x, y);
    wxMemoryDC dc;
    dc.SelectObject(bitmap);
    dc.SetBrush(wxBrush(wxT("white"), wxSOLID));
    dc.Clear();
    m_shape->Draw(dc);
    dc.SelectObject(wxNullBitmap);
    DnDShapeDataObject *self = (DnDShapeDataObject *)this;
    self->m_dobjBitmap.SetBitmap(bitmap);
    self->m_hasBitmap = true;
}
```

我们自定义的数据对象的实现到此为止就全部完成了,部分细节(比如图形怎样把自己绘制到用户界面上)没有在此列出,你可以参考wxWidgets自带的samples/dnd中的代码.

wxWidgets中的拖放相关的一些帮助

下面我们来描述一些在实现拖放操作时可以给你帮助的控件.

wxTreeCtrl

你可以使用EVT_TREE_BEGIN_DRAG或EVT_TREE_BEGIN_RDRAG事件映射宏来增加对鼠标左键或右键开始的拖放操作的处理,这是这个控件内部的鼠标事件处理函数实现的.在你的事件处理函数中,你可调用wxtreeEvent::Allow来允许 wxtreeCtrl使用它自己的拖放实现来发送一个EVT_TREE_END_DRAG事件.如果你选择了使用tree控件自己的拖放代码,那么随着拖放鼠标指针的移动,将会有一个小的拖动图片被创建,并随之移动,整个放置的操作则完全需要在应用程序的结束放置事件处理函数中实现.

下面的例子演示了怎样使用树状控件提供的拖放事件,来实现当用户把树状控件中的一个子项拖到另外一个子项上的时候,产生一个被拖动子项的拷贝.

```
BEGIN_EVENT_TABLE(MyTreeCtrl, wxTreeCtrl)
    EVT_TREE_BEGIN_DRAG(TreeTest_Ctrl, MyTreeCtrl::OnBeginDrag)
    EVT_TREE_END_DRAG(TreeTest_Ctrl, MyTreeCtrl::OnEndDrag)
END_EVENT_TABLE()
void MyTreeCtrl::OnBeginDrag(wxTreeEvent& event)
{
    // 需要显式的指明允许拖动
    if ( event.GetItem() != GetRootItem() )
    {
        m_draggedItem = event.GetItem();
        wxLogMessage(wxT("OnBeginDrag: started dragging %s"),
                      GetItemText(m_draggedItem).c_str());
        event.Allow();
    }
    else
    {
        wxLogMessage(wxT("OnBeginDrag: this item can't be dragged."));
    }
}
void MyTreeCtrl::OnEndDrag(wxTreeEvent& event)
{
    wxTreeItemId itemSrc = m_draggedItem,
                 itemDst = event.GetItem();
    m_draggedItem = (wxTreeItemId)0l;
    // 在哪里拷贝这个子项呢?
    if ( itemDst.IsOk() && !ItemHasChildren(itemDst) )
    {
        // 这种情况下拷贝到它的父项内
        itemDst = GetItemParent(itemDst);
    }
    if ( !itemDst.IsOk() )
    {
        wxLogMessage(wxT("OnEndDrag: can't drop here."));
        return;
    }
    wxString text = GetItemText(itemSrc);
    wxLogMessage(wxT("OnEndDrag: '%s' copied to '%s'."),
                 text.c_str(), GetItemText(itemDst).c_str());
    //  增加新的子项
    int image = wxGetApp().ShowImages() ? TreeCtrlIcon_File : -1;
    AppendItem(itemDst, text, image);
}
```

如果你想自己处理拖放操作,比如使用wxDropSource来实现,你可以在拖放开始事件处理函数中使用wxtreeEvent:: Allow函数来禁止默认的拖放动作,并且开始你自己的拖放动作.这种情况下拖放结束事件将不会被发送,因为你已经决定用自己的方式来处理拖放(如果使用wxDropSource::DoDragDrop函数,你需要自己检测何时拖放结束).

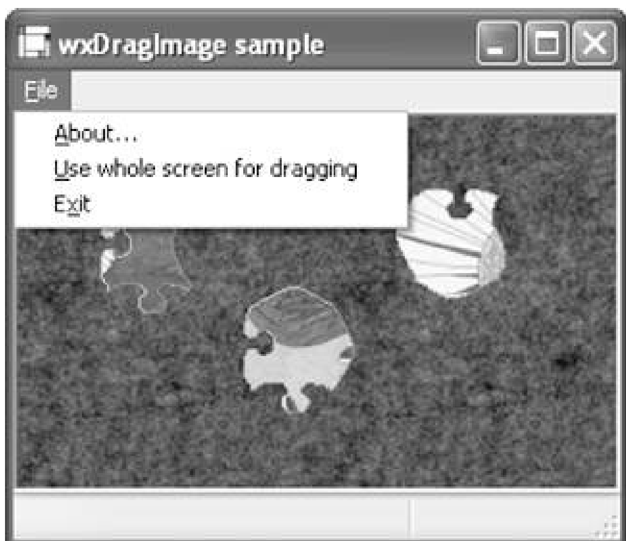
wxListCtrl

这个类没有提供默认的拖动图片,或者拖放结束事件,但是,它可以让你知道什么时候开始一个拖放操作.使用 EVT_LIST_BEGIN_DRAG或EVT_LIST_BEGIN_RDRAG事件映射宏来实现你自己的拖放代码.你也可以使用 EVT_LIST_COL_BEGIN_DRAG,EVT_LIST_COL_DRAGGING和EVT_LIST_COL_END_DRAG来检测何时某一个单独的列正在被拖动.

wxDragImage

在你实现自己的拖放操作的时候,可以很方便地使用wxDragImage类.它可以在顶层窗口上绘制一副图片,还可以移动这个图片,并且不损坏它后面的窗口.这通常是通过在移动之前保存一份背景窗口,并且在需要的时候,重绘背景窗口来实现的.

下图演示了wxDragImage例子中的主窗口,你可以在wxWidgets的samples/dragimag中找到这个例子.当主窗口上的三个拼图块被拖动时,将会采用不同的拖动图片,分别为图片本身,一个图标,或者一个动态产生的包含一串文本的图片.如果你选择使用整个屏幕这个选项,那么这个图片可以被拖动到窗口以外的地方,在Windows平台上,这个例子即可以使用标准的wxDragImage的实现(默认情形)来编译,也可以使用本地原生控件来编译,后者需要你在dragimag.cpp中将wxUSE_GENERIC_DRAGIMAGE置为1.



当检测到开始拖动操作的时候,创建一个wxDragImage对象,并且把它存放在任何你可以在整个拖动过程中访问的地方,然后调用 BeginDrag来开始拖动,调用EndDrag来结束拖动.要移动这个图片,第一次要使用Show函数,后面则需要使用Move函数.如果你需要在拖动过程当中刷新屏幕内容(比如在dragimag的例子中高量显示某个项目),你需要先调用Hide函数,然后更新你的窗口,然后调用Move函数,然后调用Show函数.

你可以在一个窗口内拖动,也可以在整个屏幕或者屏幕的任何一部分内拖动,以节省资源.如果你希望用户可以在两个拥有不同的顶层父窗口的窗口之间拖动,你就必须使用全屏拖动的方式.全屏拖动的效果不一定完美,因为它在开始拖动的时候获取了一副整个屏幕的快照,屏幕后续的改动则不会进行相应的更新.如果在你的拖动过程当中,别的应用程序对屏幕内容进行了改动,将会影响到拖动的效果.

在接下来的例子中,基于上面的那个例子,MyCanvas窗口显示了很多DragShap类的图片,它们中的每一个都和一副图片绑定. 当针对某个DragShap的拖动操作开始时,一个使用其绑定的图片的wxDragImage对象被创建,并且BeginDrag被调用.当检测到鼠标移动的时候,调用wxDragImage::Move函数来移动来将这个对象进行相应的移动.最后,当鼠标左键被释放的时候,用于指示拖动的图片被释放,被拖动的图片则在其新的位置被重绘.

```
void MyCanvas::OnMouseEvent(wxMouseEvent& event)
{
    if (event.LeftDown())
    {
        DragShape* shape = FindShape(event.GetPosition());
        if (shape)
        {
            // 我们姑且认为拖动操作已经开始
            // 不过最好等待鼠标移动一段时间再真正开始.
```

```
            m_dragMode = TEST_DRAG_START;
            m_dragStartPos = event.GetPosition();
            m_draggedShape = shape;
        }
    }
    else if (event.LeftUp() && m_dragMode != TEST_DRAG_NONE)
    {
        // 拖动操作结束
        m_dragMode = TEST_DRAG_NONE;
        if (!m_draggedShape || !m_dragImage)
            return;
        m_draggedShape->SetPosition(m_draggedShape->GetPosition()
                            + event.GetPosition() - m_dragStartPos);
        m_dragImage->Hide();
        m_dragImage->EndDrag();
        delete m_dragImage;
        m_dragImage = NULL;
        m_draggedShape->SetShow(true);
        m_draggedShape->Draw(dc);
        m_draggedShape = NULL;
    }
    else if (event.Dragging() && m_dragMode != TEST_DRAG_NONE)
    {
        if (m_dragMode == TEST_DRAG_START)
        {
            // 我们将在鼠标已经移动了一小段距离以后开始真正的拖动
            int tolerance = 2;
            int dx = abs(event.GetPosition().x - m_dragStartPos.x);
            int dy = abs(event.GetPosition().y - m_dragStartPos.y);
            if (dx &lt;= tolerance && dy &lt;= tolerance)
                return;
            // 开始拖动.
            m_dragMode = TEST_DRAG_DRAGGING;
            if (m_dragImage)
                delete m_dragImage;
            // 从画布上清除拖动图片
            m_draggedShape->SetShow(false);
            wxClientDC dc(this);
            EraseShape(m_draggedShape, dc);
            DrawShapes(dc);
            m_dragImage = new wxDragImage(
                                   m_draggedShape->GetBitmap());
            // 被拖动图片的左上角到目前位置的偏移量
            wxPoint beginDragHotSpot = m_dragStartPos
                                 m_draggedShape->GetPosition();
            // 总认为坐标系为被捕获窗口的客户区坐标系
            if (!m_dragImage->BeginDrag(beginDragHotSpot, this))
            {
                delete m_dragImage;
                m_dragImage = NULL;
                m_dragMode = TEST_DRAG_NONE;
            } else
            {
                m_dragImage->Move(event.GetPosition());
                m_dragImage->Show();
            }
        }
        else if (m_dragMode == TEST_DRAG_DRAGGING)
        {
            // 移动这个图片
            m_dragImage->Move(event.GetPosition());
        }
    }
}
```

如果你希望自己绘制用于拖动的图片而不是使用一个位图,你可以实现一个wxGenericDragImage的派生类,重载其 wxDragImage::DoDrawImage函数和wxDragImage::GetImageRect函数.在非windows的平台上, wxDragImage是

wxGenericDragImage的一个别名而已,而windows平台上实现的wxDragImage不支持DoDrawImage函数,也限制只能绘制有时候显得有点恐怖的半透明图片,因此,你可以考虑在所有的平台上都使用 wxGenericDragImage类.

当你开始拖动操作的时候,就在正准备调用wxDragImage::Show函数之前,通常你需要现在屏幕上擦除你要拖动的对象,这可以使得 wxDragImage保存的背景中没有正在拖动的对象,因此整个的拖动过程看上去也更合理,不过这将导致屏幕在开始拖动的时候会有一点点的闪烁.要避免这种闪烁(仅适用于使用wxGenericDragImage的情况),你可以重载wxGenericDragImage的UpdateBackingFromWindow函数,使用传递给你的设备上下文绘制一个不包含正在拖动对象的背景,然后你就不需要在调用 wxDragImage::Show函数之前擦除你要拖动的对象了,整个拖动过程的屏幕就将会是平滑而无闪烁的了.

第十一章小结

在这一章里,我们看到了怎样将数据传输到剪贴板上或者怎样从剪贴板获取数据.我们也了解了怎样从拖放源的角度以及拖放目的的角度实现拖放操作.还了解了 wxWidgets中和拖放相关的一些其他领域的知识.更深入的了解请参考wxWidgets的samples/dnd, samples/dragimag和samples/treectrl目录中的例子.

在下一章里,我们将回到窗口类相关的主题,介绍一些高级的窗口类以及怎样通过它们让你的应用程序进入一个更新的层级.

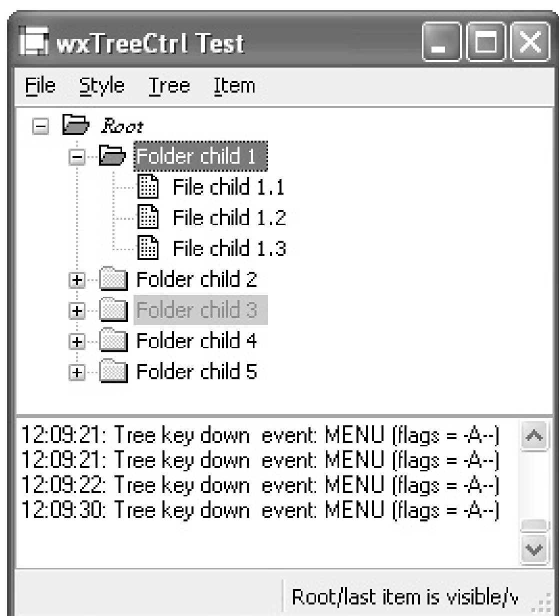
第十二章高级窗口控件

显然我们不能在这里列举wxWidgets提供的所有的控件,但是还是有必要对其中几个更高级一点的控件作一些介绍,以便在需要的时候你可以更好的使用它们.本章覆盖的内容包括下面的主题:

- wxTReeCtrl; 这个控件用来帮助你为分等级的数据建模.
- wxListCtrl; 这个控件让你以灵活的方式显示一组文本标签和图标.
- wxWizard; 这个控件使用多个页面对某个特定的任务提供向导机制.
- wxHtmlWindow; 你可以在"关于"对话框和报告对话框(以及其他你可以想到的对话框)中使用的轻量级的HTML显示控件.
- wxGrid; 网格控件用是一个支持多种特性的标状数据显示控件.
- wxTaskBarIcon; 这个控件让你的程序可以很容易的访问系统托盘区或者类似的区域.
- 编写自定义的控件. 介绍了制作一个专业级的自定义控件必须的几个步骤

12.1 wxTreeCtrl

树状控件以层的形式展示信息,它的子项可以展开也可以合并.下图演示了wxWidgets的树状控件例子,它正以不同的字体和风格以及颜色进行展示.每一个树状控件的子项都代表一个wxtreeItemId对象,它拥有一个文本标签和一个可选图标,并且文本和图标的内容都可以动态修改.树状控件可以以单选或者多选的形式创建.如果你希望在wxtreeItemId上绑定一些数据,你需要实现自己的wxTreeItemData派生类,然后调用 wxTreeCtrl::SetItemData函数以及wxTreeCtrl::GetItemData函数.这个数据在子项被释放的时候将会被一并释放(delete调用),如果你将其指向你实际的数据,需要注意避免重复释放.



因为应用程序可以检测到树状控件的子项被单击的事件,你可以用这个特点通过更改子项的图片来达到模拟其他的控件的目的.比如说,你可以很容易用树状控件的子项来模拟一个复选框.

下面的代码演示了怎样创建一个树状控件,定义其子项的绑定数据以及图片:

```
#include "wx/treectrl.h"
// 声明一个代表和子项绑定的数据的类
class MyTreeItemData : public wxTreeItemData
{
public:
    MyTreeItemData(const wxString& desc) : m_desc(desc) { }
    const wxString& GetDesc() const { return m_desc; }
private:
    wxString m_desc;
};
// 子项相关的图片
#include "file.xpm"
#include "folder.xpm"
// 创建一个树状控件
wxTreeCtrl* treeCtrl = new wxTreeCtrl(
    this, wxID_ANY, wxPoint(0, 0), wxSize(400, 400),
    wxTR_HAS_BUTTONS|wxTR_SINGLE);
wxImageList* imageList = new wxImageList(16, 16);
imageList->Add(wxIcon(folder_xpm);
imageList->Add(wxIcon(file_xpm);
treeCtrl->AssignImageList(imageList);
// 根节点使用文件夹图标,而两个字节点使用文件图标
wxTreeItemId rootId = treeCtrl->AddRoot(wxT("Root"), 0, 0,
                          new MyTreeItemData(wxT("Root item")));
wxTreeItemId itemId1 = treeCtrl->AppendItem(rootId,
                          wxT("File 1"), 1, 1,
                          new MyTreeItemData(wxT("File item 1")));
wxTreeItemId itemId2 = treeCtrl->AppendItem(rootId,
                          wxT("File 2"), 1, 1,
                          new MyTreeItemData(wxT("File item 2")));
```

wxTreeCtrl的窗口类型

wxTreeCtrl有如下表所示的额外的窗口类型:

wxtr_DEFAULT_STYLE	这个值是各个平台上树状控件实现和默认值最接近的值
wxtr_EDIT_LABELS	是否子项文本可编辑
wxtr_NO_BUTTONS	不必显示用于展开或者合并子项的按钮
wxtr_HAS_BUTTONS	显示用于展开或者合并子项的按钮
wxTR_NO_LINES	不必显示用于表示层级关系的垂直虚线
wxtr_FULL_ROW_HIGHLIGHT	当选中某个子项的时候高亮显示整行(在windows平台上,除非设置了wxtr_NO_LINES,否则这个类型将被忽略)
wxtr_LINES_AT_ROOT	不必显示根节点之间的连线.这个类型只有在设置wxtr_HIDE_ROOT 并且没有设置wxtr_NO_LINES 的时候有效
wxtr_HIDE_ROOT	不显示根节点,这将导致第一层的字节点成为一系列根节点
wxtr_ROW_LINES	使用这个类型在已显示的行之间绘制一个高对比的边界
wxTR_HAS_VARIABLE_ROW_HEIGHT	设置这个类型允许各行采用不同的高度,否则各行都将采用和最大的行高同样的高度.这个类仅适用于树状控件的标准实现(而非各个平台的原生实现)
wxtr_SINGLE	单选模式
wxtr_MULTIPLE	多选模式
wxtr_EXTENDED	允许多选非连续的子项(该功能仅是部分实现)

wxTreeCtrl的事件

树状控件产生wxtreeEvent类型的事件,这种事件可以在父子关系的窗口之间传递.

EVT_TREE_BEGIN_DRAG(id, func)EVT_TREE_BEGIN_RDRAG(id, func)	在用户开始拖放操作的时候产生,这个事件的使用细节请参考第**11**章,**"**剪贴板和拖放操作**"**
EVT_TREE_BEGIN_LABEL_EDIT(id, func) EVT_TREE_END_LABEL_EDIT(id, func)	当用户开始编辑或者刚刚完成编辑子项标签的时候产生
EVT_TREE_DELETE_ITEM(id, func)	当某个子项被删除的时候产生
EVT_TREE_GET_INFO(id, func)	当某个子项的数据被请求的时候产生
EVT_TREE_SET_INFO(id, func)	当某个子项的数据被设置的时候产生
EVT_TREE_ITEM_ACTIVATED(id, func)	当某个子项被激活(双击或者使用键盘选择)的时候产生
EVT_TREE_ITEM_COLLAPSED(id, func)	给定的子项已被收缩(合并)的时候产生
EVT_trEE_ITEM_COLLAPSING(id, func)	给定的子项即将收缩(合并)的时候产生,这个事件可以被Veto以阻止收缩.
EVT_TREE_ITEM_EXPANDED(id, func)	给定子项已被展开的时候产生
EVT_TREE_ITEM_EXPANDING(id, func)	给定子项即将展开的时候产生,这个事件可以被Veto以阻止展开
EVT_TREE_SEL_CHANGED(id, func)	选中的子项发生变化以后(新的子项被选中或者旧的选中项不被选中的时候)产生
EVT_TREE_SEL_CHANGING(id, func)	选中的子项即将发生变化的时候产生,该事件可以被Veto以阻止变化产生
EVT_TREE_KEY_DOWN(id, func)	检测针对该树状控件的键盘事件
EVT_TREE_ITEM_GET_TOOLTIP(id, func)	这个事件仅支持windows平台,它使得你可以给某个子项设置单独的工具提示

wxTreeCtrl的成员函数

下面列出了wxTreeCtrl控件的一些重要的成员函数.

使用AddRoot函数增加第一个子项,然后使用AppendItem, InsertItem或PrependItem来增加随后的子项.使用Delete移除一个子项,使用DeleteAllItems删除某个子项所有的子项,或者使用DeleteChildren删除某个子项的所有直接子项.

使用SetItemText设置某个子项的标签,使用SetItemTextColour,SetItemBackgroundColour,SetItemBold和SetItemFont来设置标签的外观.

如果你想给某个子项指定一幅图片,首先需要使用SetImageList函数将某个图片列表和这个树状控件绑定.每个子项可以指定四个状态的图片,分别是wxTReeItemIcon_Normal,wxTReeItemIcon_Selected, wxtreeItemIcon_Expanded和

wxTReeItemIcon_SelectedExpanded,你可以使用 SetItemImage函数给每个状态指定一个图片列表中图片索引.如果你只给wxTReeItemIcon_Normal状态指定了一个索引,那么别的状态也将都使用这个图片.

使用Scroll函数以便将某个子项移动到可见区域,使用EnsureVisible使得这个子项在需要的时候展开以便其可以位于可见区域.使用Expand函数展开某个子项,Collapse和CollapseAndReset函数合并某个子项,后者还将移除其所有的子项,如果你正在使用的树状控件有很多子项,你可能希望只增加可见部分的子项以便提高性能.在这种情况下,你可以处理EVT_TREE_ITEM_EXPANDING事件,在需要的时候才增加子项,在收缩的时候则移除所有子项.而且你还需要调用SetItemHasChildren函数以便没有子项的子项也可以显示一个可扩展按钮,即使它真的没有.

使用SelectItem选择或者去选择某个子项.如果是单选类型,你可以使用GetSelection函数得到正被选中的子项,如果当前没有子项被选中,则返回一个未初始化的wxTReeItemId,你可以调用wxTreeItemId::IsOk函数来判断其有效性.而对于多选类型,你可以使用GetSelections函数获取当前选中的子项,你需要传递一个wxArrayTreeItemItemIds类型的引用作为参数. Unselect函数在单选情况下去选中当前的子项,而UnselectAll函数则用在多选情况下去选中所有正被选中的子项,UnselectItem 函数可以用来在多选情况下去选中某一个子项.

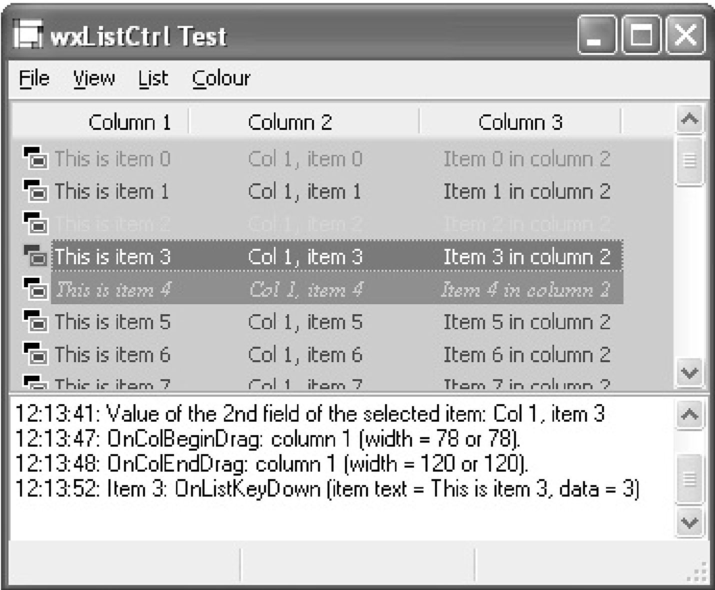
遍历某个树状控件的所有子项也有多种方法:你可以先使用GetRootItem函数获得根节点,然后使用GetFirstChild和 GetNextChild遍历所有子项.使用GetNextSibling和GetPrevSibling获取某个子项后一个和前一个兄弟节点.使用 ItemHasChildren函数判断某个子项是否有字节点,使用GetParent函数获取某个子项的父节点.GetCount函数则用来返回树状控件中所有子项的个数,而GetChildrenCount则返回某个子项的字节点的数目.

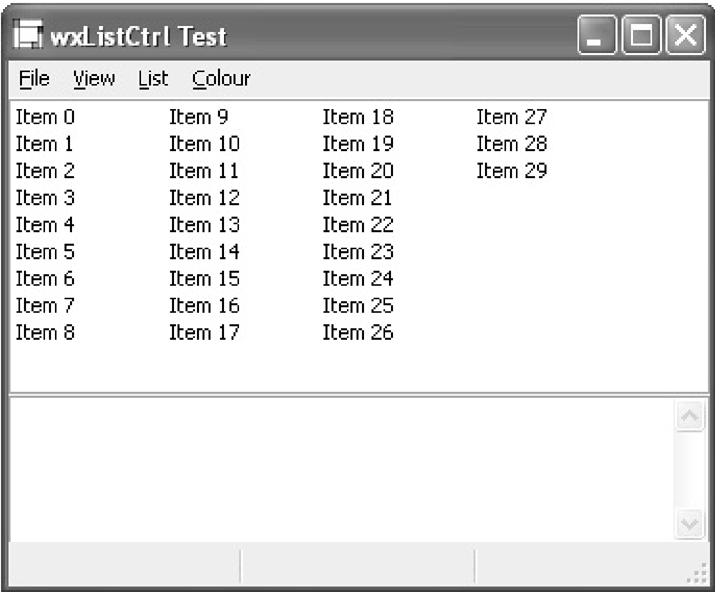
HitTest函数在实现你自己拖放的时候是很有用的,它使得你可以通过鼠标位置找到这个位置对应的子项以及子项的某个特定部分.具体返回值请参考相关手册中的内容.使用GetBoundingRect函数可以得到某个子项对应的矩形区域.

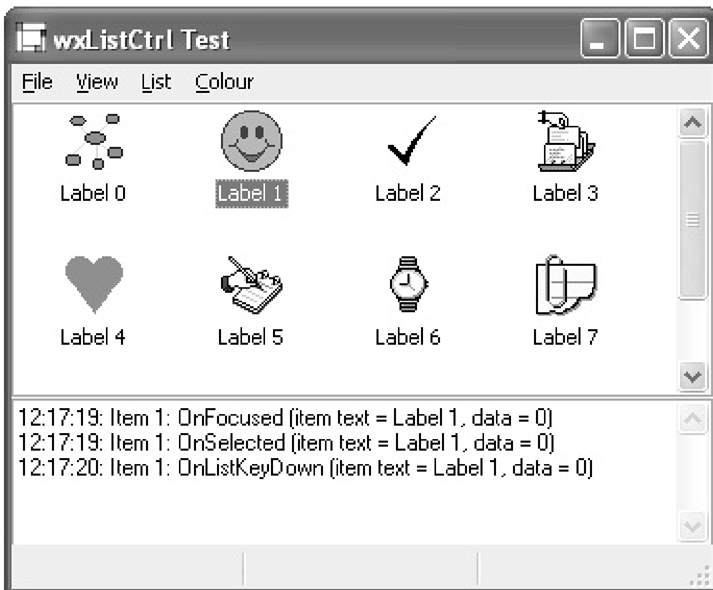
更多关于树状控件的信息请参考使用手册以及samples/treectrl中的wxTreeCtrl例子.

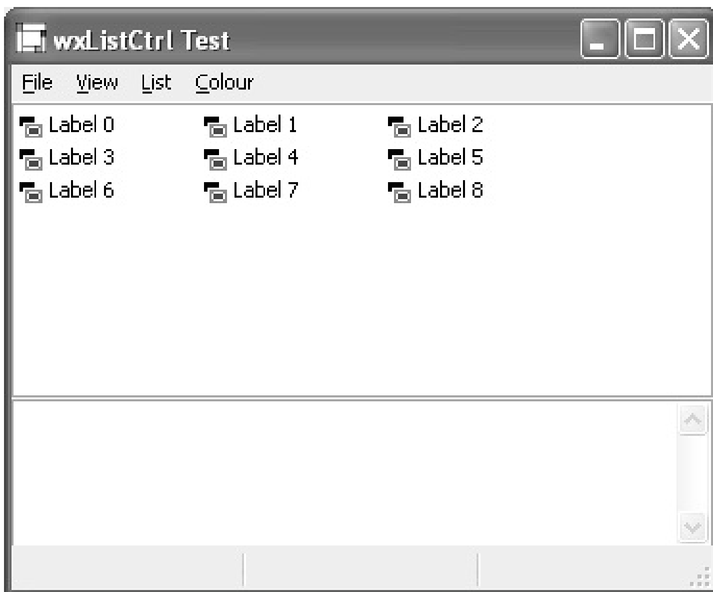
12.2 wxListCtrl

列表控件使用四种形式中的一种来显示子项:多列视图,多列报告视图,大图标方式以及小图标方式.下图分别对齐进行了演示.每一个子项使用一个长整型的索引来表示的,随着子项的增加,删除以及排序,这个索引可能发生变化.和树状控件不同,列表框默认采用允许多选的方式,不过你还是可以在创建窗口的时候通过类型指定只允许单选.如果你需要对所有子项进行排序,你可以提供一个排序函数.在多列报告视图中,可以给每一列增加一个标题,点过拦截标题单击事件可以实现一些附加操作比如按当前列进行排序.每一列的宽度既可以通过代码改变,也可以通过用户使用鼠标拖拽来改变.









列表控件的每个子项同样可以绑定一些客户区数据,但是和树状控件不同,每一个列表框的子项只能绑定一个长整型数据.如果你希望给每一个子项绑定一个特定对象,你需要自己实现一个从长整型到特定数据对象之间的隐射,并且自己负责那些数据对象的创建和释放.

wxListCtrl的窗口类型

wxLC_LIST	使用可选的小图标进行多列显示.列数是自动计算的,不需要设置象**wxLC_REPORT**那样设置列数,换句话说,这只是一个自动换行的排列.
wxLC_REPORT	单列或者多列报告方式,并且可以设置可选的标题.
wxLC_VIRTUAL	指定显示的文本由应用程序动态提供; 只能用于wxLC_REPORT方式.
wxLC_ICON	大图标方式显示,可选显示文本标签.
wxLC_SMALL_ICON	小图标方式显示,可选显示文本标签
wxLC_ALIGN_TOP	图标顶端对齐. 仅适用于Windows.
wxLC_ALIGN_LEFT	图标左对齐.
wxLC_AUTO_ARRANGE	图标自动排列.仅适用于Windows.
wxLC_EDIT_LABELS	标签可编辑; 当编辑动作开始时应用程序将收到通知.
wxLC_NO_HEADER	在报告模式下不显示标题.
wxLC_SINGLE_SEL	指定单选模式; 默认为多选模式.
wxLC_SORT_ASCENDING	从小到大排序. 应用程序需要在SortItems中提供自己的排序函数.
wxLC_SORT_DESCENDING	从大到小排序. 应用程序需要在SortItems中提供自己的排序函数.
wxLC_HRULES	在报告模式中显示每行之间的标尺.
wxLC_VRULES	在报告模式中显示每列之间的标尺.

wxListCtrl事件

wxListCtrl产生wxListEvent类型的事件,如下表所示,事件可以父子窗口之间传递,wxListEvent::GetIndex可以用来返回针对单个子项的事件中的子项索引.

EVT_LIST_BEGIN_DRAG(id, func) EVT_LIST_BEGIN_RDRAG(id, func)	这个事件在拖放开始的时候产生,如果要实现拖放,你需要提供拖放操作的剩下的部分.通过**wxListEvent::GetPoint**函数获取当前的鼠标位置.
EVT_LIST_BEGIN_LABEL_EDIT(id, func) EVT_LIST_END_LABEL_EDIT(id, func)	用户正准备编辑标签或者结束编辑的时候产生,使用Veto函数可禁止这种编辑行为.wxListEvent::GetText则返回当前的标签.
EVT_LIST_DELETE_ITEM(id, func) EVT_LIST_DELETE_ALL_ITEMS(id, func)	事件在某个子项被删除或者所有的子项都被删除的时候产生.
EVT_LIST_ITEM_SELECTED(id, func) EVT_LIST_ITEM_DESELECTED(id, func)	在子项的选中状态发生改变的时候产生.
EVT_LIST_ITEM_ACTIVATED(id, func)	在某个子项被激活(鼠标双击或者通过键盘)的时候产生.
EVT_LIST_ITEM_FOCUSED(id, func)	当当前焦点子项发生改变的时候产生.
EVT_LIST_ITEM_MIDDLE_CLICK(id, func) EVT_LIST_ITEM_RIGHT_CLICK(id, func)	在子项被鼠标以中键或者右键单击的时候产生.
EVT_LIST_KEY_DOWN(id, func)	在有针对列表控件的按键事件的时候产生. 使用wxListEvent::GetKeyCode函数来得到当前按键的编码.
EVT_LIST_INSERT_ITEM(id, func)	新的子项插入的时候产生.
EVT_LIST_COL_CLICK(id, func) EVT_LIST_COL_RIGHT_CLICK(id, func)	某一列被单击的时候产生.使用wxListEvent::GetColumn函数获得被单击的列索引.
EVT_LIST_COL_BEGIN_DRAG(id, func) EVT_LIST_COL_DRAGGING(id, func) EVT_LIST_COL_END_DRAG(id, func)	列大小发生改变的时候或者结束改变的时候产生.你可以使用Veto函数来禁止这种改变.使用wxListEvent::GetColumn获得关联的列索引.
EVT_LIST_CACHE_HINT(id, func)	如果你正在实现一个续列表控件,你可能想在某一组子项被显示之前更新其内部数据.这个事件通知你进行这个操作的最合适的时机.使用wxListEvent::GetCacheFrom函数和wxListEvent::GetCacheTo函数来获得即将更新的子项的索引范围.

wxListItem

你需要使用wxListItem这个类在列表控件中进行插入,设置子项属性或者获取子项属性的操作.

SetMask函数用来指示你希望使用的列表子项属性,如下表所示:

wxLIST_MASK_STATE	**state**属性有效.
wxLIST_MASK_TEXT	text属性有效.
wxLIST_MASK_IMAGE	image属性有效.
wxLIST_MASK_DATA	data属性有效.
wxLIST_MASK_WIDTH	width属性有效.
wxLIST_MASK_FORMAT	format属性有效.

使用SetId函数设置子项基于0的索引值,使用SetColumn函数设置当控件在报告模式时基于0的列索引.

SetState函数则用来设置下表所示的子项状态:

wxLIST_STATE_DONTCARE	不关心子项状态.
wxLIST_STATE_DROPHILITED	子项正被高亮显示以便接受一个拖放中的放置事件(仅适用于Windows).
wxLIST_STATE_FOCUSED	子项正拥有焦点.
wxLIST_STATE_SELECTED	子项正被显示.
wxLIST_STATE_CUT	子项已被剪切(仅适用于Windows).

SetStateMask则用来设置当前正在更改的状态,参数和上表中的值相同.

SetText函数用来设置标签或者标题文本.SetImage函数用来设置子项的图片在图片列表中的索引.

SetData函数则用来设置和子项绑定的长整型客户区数据.

SetFormat函数用来设置列表模式时的对齐模式,其值为wxLIST_FORMAT_LEFT, wxLIST_FORMAT_RIGHT或wxLIST_FORMAT_CENTRE(或者wxLIST_FORMAT_CENTER), SetColumnWidth函数则可以用来设置列宽.

还有一些其它的可视属性可以通过下面的这些函数设置:SetAlign, SetBackgroundColour, SetTextColour和SetFont.这些属性不需要设置掩码标记.并且所有Set函数都拥有一个对应的Get函数来获取相应的设置.

下面的代码用wxListItem来选择第二个子项,并且设置其文本标签以及字体颜色.

```
wxListItem item;
item.SetId(1);
item.SetMask(wxLIST_MASK_STATE|wxLIST_MASK_TEXT);
item.SetStateMask(wxLIST_STATE_SELECTED);
item.SetState(wxLIST_STATE_SELECTED);
item.SetTextColour(*wxRED);
item.SetText(wxT("Red thing"));
listCtrl->SetItem(item);
```

你也可以直接通过另外的函数来设置这些属性.比如SetItemText, SetItemImage, SetItemState, GetItemText, GetItemImage, GetItemState等等,我们马上就会谈到这些函数.

wxListCtrl成员函数

在windows平台上可以调用Arrange函数在大图标或者小图标的方式下进行排列图标.

AssignImageList给列表控件绑定一个图标列表,图形列表的释放由列表控件负责,SetImageList也可以实现同样的功能,不过图片列表的释放由应用程序自己负责,使用wxIMAGE_LIST_NORMAL或wxIMAGE_LIST_SMALL来指定用于哪种显示形式,GetImageList则用来获取对应的图片列表指针.

InsertItem函数用来在控件的特定位置中插入一个子项.可以传递的参数包括几种形式:直接传递已经设置了成员变量的 wxListItem对象.或者一个索引和一个字符串,一个索引和一个图片索引以及一个索引,一个标签和一个图片索引等. InsertColumn函数则用来在报告视图中插入一列.

ClearAll函数删除所有的子项以及报告视图中的列信息,并产生一个所有子项被删除的事件.DeleteAllItems删除所有的子项并产生所有的子项被删除的事件.DeleteItem删除某一个子项并产生子项被删除事件.DeleteColumn则用来删除报告视图中的某一列.

SetItem函数通过传递一个wxListItem变量来设置某个子项的属性,就象前面例子中的那样,或者你还可以传递一个索引,一个列,一个标签以及图片索引参数.GetItem则用来查询那些经由wxListItem::SetId函数设置的子项索引的信息.

SetItemData函数用来给某个子项绑定一个长整数.在绝大多数的平台上,都可以将一个指针强制转换成长整数,以便这里可以设置一个指针,但是某些平台上,这样作是不适合的,这时候你可以通过一个哈希映射来将一个长整数和某个对象对应.GetItemData则返回某个子项绑定的长整数.需要注意的是,子项的位置随着其它子项的插入或者删除以及排序操作,将会发生改变,因此不适合用来索引一个子项,但是子项绑定的客户数据是不会随子项位置的变化而变化的,可以用来唯一标识一个子项.

SetItemImage函数采用子项索引和图片索引两个参数给某个子项指定一个图标.

SetItemState用来设置子项的某个状态值,必须通过一个掩码来指定哪个状态的值将改变,参考前面介绍wxListItem的部分.GetItemState则用来返回某个给定子项的状态.

SetItemText函数和GetItemText函数用来设置和获取某个子项的文本标签.

SetTextColour用来设置所有子项的文本标签的颜色,SetBackgroundColour则用来设置控件的背景颜色. SetItemTextColour和SetItemBackgroundColour用来设置在报告模式下某个单独子项的文本前景色和背景色.以上函数都由对应的Get函数.

使用EditLabel函数开始编辑某个子项的标签,这将会导致一个wxEVT_LIST_BEGIN_LABEL_EDIT事件.你可以通过这个事件的GetEditControl函数获得对应编辑控件的指针(仅适用于windows).

EnsureVisible将使得指定的子项处于可见区域.ScrollList用来在某个方向上滚动指定的象素(仅适用于 windows),RefreshItem用来刷新某个子项的显示,RefreshItems则用来刷新一系列子项的显示,这两个函数主要用在使用虚列表控件而子项的内部数据已经改变时.GetTopItem用于在列表或者报告视图中返回第一个可见的子项.

FindItem可以被用来查找指定标签,客户区数据或者位置的子项.GetNextItem则用来查找某些指定状态的子项(比如所有正被选中的子项).HitTest函数用来返回给定位置的子项.GetItemPosition返回大图标或者小图标视图中某个子项的相对于客户区的起始座标.GetItemRect则返回相对于客户区的座标以及子项所占的大小.

你可以动态改变列表控件的显示类型而不需要先释放再重新创建它:使用SetSingleStyle函数指定一个类型,比如wxLC_REPORT.指定false参数来移除某一个显示类型.

在报告模式中使用SetColumn函数来设置某列的信息,比如标题或者宽度,参考前面的wxListItem的相关介绍,作为一个简单的替代版本,SetColumnWidth用来直接设置某个列的宽度.使用GetColumn和GetColumnWidth来获取对应的设置.而 GetColumnCount则用来获取当前的列数.

GetItemCount函数用来返回列表控件中子项的数目,GetSelectedItemCount函数则用来返回选中的子项的数目.GetCountPerPage函数在大小图标方式中返回所有子项的数目,而在列表和报告模式中返回可见区域可以容纳的子项的数目.

最后,SortItems函数可以被用来对列表控件中的子项进行排序.这个函数的参数为比较函数wxListCtrlCompare的地址,这个比较函数使用的参数包括两个子项对应的客户区数据,另外一个更深入的整数,然会另外一个整数,如果两个子项比较的结果为相等,则返回0,前一个大于后一个则返回正数,前一个小于后一个则返回负数.当然,为了排序功能更好的工作,你需要给每个子项指定一个长整型的客户区数据(可以通过 wxListItem::SetData函数).这些客户区数据被传递给比较函数以实现比较.

使用wxListCtrl

下面的代码创建和显示了一个报告模式的列表控件,这个列表控件公有三列10个子项,每一行开始的地方都会显示一个16x16的文件图标:

```
#include "wx/listctrl.h"
// 报告模式中的图片
#include "file.xpm"
#include "folder.xpm"
// 创建一个报告模式的列表控件
wxListCtrl* listCtrlReport = new wxListCtrl(
    this, wxID_ANY, wxDefaultPosition, wxSize(400, 400),
    wxLC_REPORT|wxLC_SINGLE_SEL);
// 绑定一个图片列表
wxImageList* imageList = new wxImageList(16, 16);
imageList->Add(wxIcon(folder_xpm));
imageList->Add(wxIcon(file_xpm));
listCtrlReport->AssignImageList(imageList, wxIMAGE_LIST_SMALL);
// 插入三列
wxListItem itemCol;
itemCol.SetText(wxT("Column 1"));
itemCol.SetImage(-1);
listCtrlReport->InsertColumn(0, itemCol);
listCtrlReport->SetColumnWidth(0, wxLIST_AUTOSIZE );
itemCol.SetText(wxT("Column 2"));
itemCol.SetAlign(wxLIST_FORMAT_CENTRE);
listCtrlReport->InsertColumn(1, itemCol);
listCtrlReport->SetColumnWidth(1, wxLIST_AUTOSIZE );
itemCol.SetText(wxT("Column 3"));
itemCol.SetAlign(wxLIST_FORMAT_RIGHT);
listCtrlReport->InsertColumn(2, itemCol);
listCtrlReport->SetColumnWidth(2, wxLIST_AUTOSIZE );
// 插入10个子项
for ( int i = 0; i &lt; 10; i++ )
{
    int imageIndex = 0;
    wxString buf;
    // 插入一个子项,字符串用于第1栏,
    // 图片索引为0
    buf.Printf(wxT("This is item %d"), i);
    listCtrlReport->InsertItem(i, buf, imageIndex);
    // 子项可能由于各种原因(比如:排序)改变索引,
    // 因此将现在的索引保存为客户区数据
    listCtrlReport->SetItemData(i, i);
    // 为第2栏设置一个文本
    buf.Printf(wxT("Col 1, item %d"), i);
    listCtrlReport->SetItem(i, 1, buf);
    // 为第三栏设置一个文本
    buf.Printf(wxT("Item %d in column 2"), i);
    listCtrlReport->SetItem(i, 2, buf);
}
```

虚列表控件

通常,列表控件自己保存所有子项相关的文本标签,图片以及其它可见属性的信息,对于小量数据来说,这是不成问题的,但是如果你有成千上万的子项,你可能需要实现一个虚的列表控件. 虚列表控件由应用程序保存子项的数据,你需要重载它的三个虚函数OnGetItemLabel,OnGetItemImage和 OnGetItemAttr,列表控件会在需要的时候调用它们.你必须调用SetItemCount函数来设置当前的子项个数,因为你将不会实际增加任何子项.你还可以处理EVT_LIST_CACHE_HINT事件以便在某一部分子项即将被显示直接更新相关的内部数据,下面是重载三个函数的一个简单的例子:

```
wxString MyListCtrl::OnGetItemText(long item, long column) const
{
    return wxString::Format(wxT("Column %ld of item %ld"), column, item);
}
int MyListCtrl::OnGetItemImage(long WXUNUSED(item)) const
{
    // 全部的子项都返回0号索引的图片
    return 0;
}
wxListItemAttr *MyListCtrl::OnGetItemAttr(long item) const
{
    // 这个成员是内部使用用来为每一个子项保存相关属性信息
    return item % 2 ? NULL : (wxListItemAttr *)&m_attr;
}
```

下面演示怎样创建一个虚列表控件,我们不用增加任何子项,并且故意将它的子项个数设置为一个略显夸张的值:

```
virtualListCtrl = new MyListCtrl(parent, wxID_ANY,
      wxDefaultPosition, wxDefaultSize, wxLC_REPORT|wxLC_VIRTUAL);
virtualListCtrl->SetImageList(imageListSmall, wxIMAGE_LIST_SMALL);
virtualListCtrl->InsertColumn(0, wxT("First Column"));
virtualListCtrl->InsertColumn(1, wxT("Second Column"));
virtualListCtrl->SetColumnWidth(0, 150);
virtualListCtrl->SetColumnWidth(1, 150);
virtualListCtrl->SetItemCount(1000000);
```

当控件内部数据改变时,如果子项总数改变,你需要重新设置其数目,然后调用wxListCtrl::RefreshItem或者wxListCtrl::RefreshItems来刷新子项的显示.

参考samples/listctrl目录中的完整的例子.

12.3 wxWizard

使用向导是将一堆复杂的选项变成一系列相对简单的对话框的一个好方法.它通常用来帮助应用程序的使用新手们开始使用某个特定的功能,比如,搜集建立新工程需要的信息,导出数据等等.向导中的选项通常都在应用程序别的用户界面上有体现,但是提供一个向导以便用户可以专注于那些为了完成某个任务必须要设置的选项.

向导控件通常在一个窗口内提供一系列的类似对话框的窗口,这些窗口的左边都拥有一副图片(可以是一样的也可以是不一样的),而底部则通常用一系列导航按钮,随着用户的选择进入预先设置好的下一页面,下一页面的索引是随用户的选择而变化的,因此不是所有的页面都会在一次向导执行过程中全部显示.

当标准的向导导航按钮被按下时,将会产生对应的事件,向导类或者其派生类可以捕获相应的事件.

要显示一个向导,你需要先创建一个向导(或者其派生类),然后创建子页面(作为向导的子窗口).你可以使用 wxWizardPageSimple类(或其派生类)来创建向导页面,然后使用wxWizardPageSimple::Chain函数将其和一个向导绑定.或者,如果你需要动态调整页面的顺序,你可以使用wxWizardPage的派生类,重载其GetPrev和GetNext函数. 然后将每一个页面增加到GetPageAreaSizer返回的布局控件中,以便向导控件自动将自己的大小调整到最大页面的大小.

向导控件只额外定义了wxWIZARD_EX_HELPBUTTON扩展类型,以便在标准向导导航按钮中增加帮助按钮.注意这是一个扩展类型,需要在Create之前使用SetExtraStyle来进行设置.

wxWizard事件

wxWizard产生wxWizardEvent类型的事件,如下表所示,这些事件将首先被发送到当前页面,如果页面没有定义处理函数, 则发送到向导本身.除了EVT_WIZARD_FINISHED事件以外,别的事件都可以调用wxWizardEvent::GetPage函数返回当前页面.

EVT_WIZARD_PAGE_CHANGED(id, func)	当导航页面已发生变化的时候产生,使用**wxWizardEvent::GetDirection**函数判断方向**(true**为向前**).**
EVT_WIZARD_PAGE_CHANGING(id, func)	当导航页面即将变化的时候产生,这个事件可以被Veto以便取消这个事件.同样可以使用wxWizardEvent::GetDirection判断方向(true为向前).
EVT_WIZARD_CANCEL(id, func)	用户点击取消按钮的时候产生,这个事件可以被Veto(使得事件导致的操作无效).
EVT_WIZARD_HELP(id, func)	用户点击帮助按钮的时候产生.
EVT_WIZARD_FINISHED(id, func)	用户点击完成按钮的时候产生.这个事件产生的时间是在向导对话框刚刚关闭以后.

wxWizard的成员函数

GetPageAreaSizer函数返回用户管理所有页面的布局控件.你需要将所有的页面增加到这个布局控件中,或者将某一个可以通过 GetNext函数访问到其它所有页面的页面增加到布局控件中,以便向导控件可以知道最大的页面的大小.如果你没有这样作,你需要在显示向导时在第一个页面显示之前调用其FitToPage函数,如果wxWizardPage::GetNext不能访问到所有的页面,你需要对每个页面调用 FitToPage函数.

GetCurrentPage函数返回当前活动的页面,如果RunWizard函数还没有被执行则返回NULL.

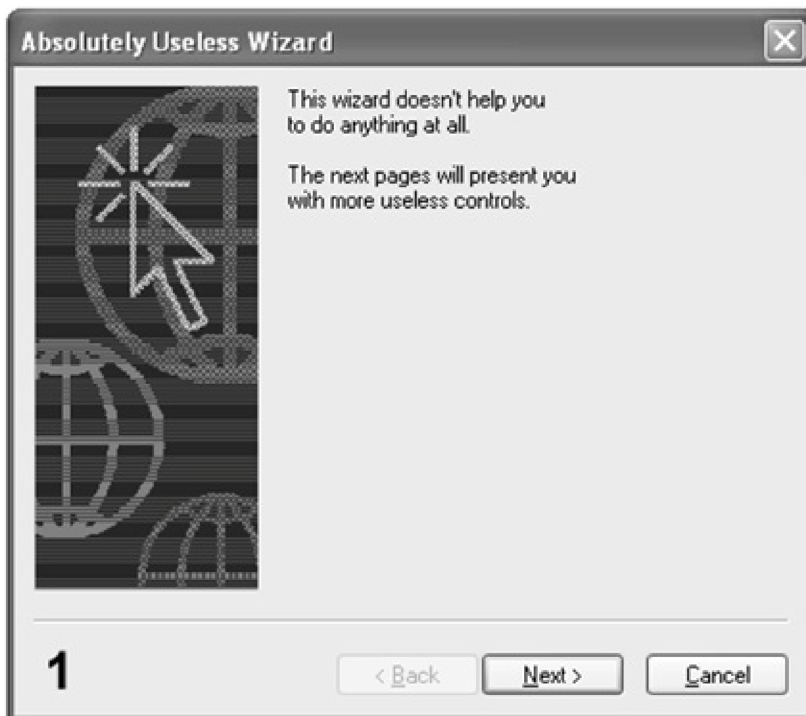
GetPageSize当前设置的页面大小. SetPageSize则用来设置所有页面使用的页面大小,不过最好还是将页面增加到GetPageAreaSizer布局控件中来决定页面大小比较合适.

调用RunWizard,传递要显示的第一个页面作为参数,以便将向导置于执行状态.如果向导执行成功这个函数返回True,如果用户取消了向导则返回False.

可以用SetBorder函数设置向导边界的大小,默认为0.

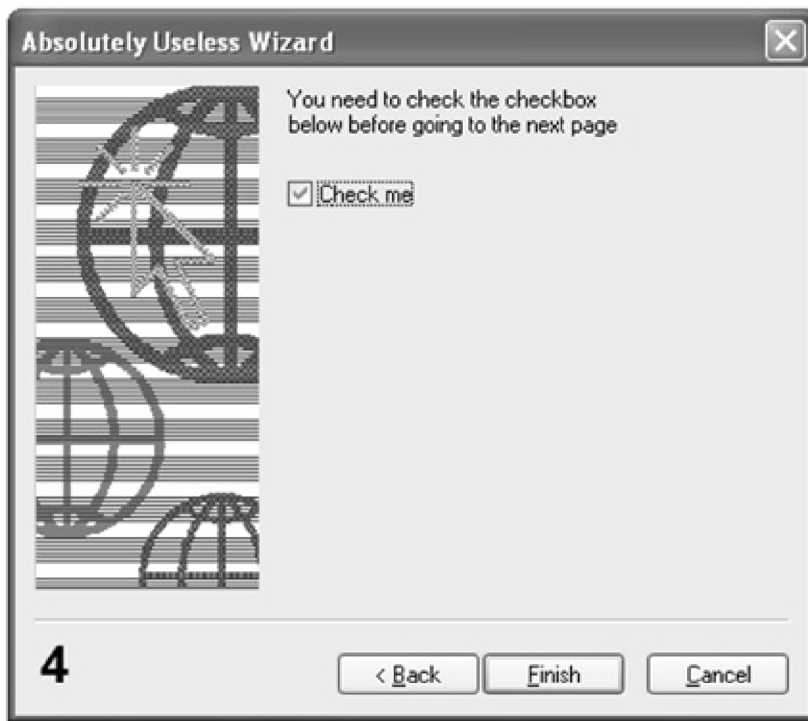
wxWizard使用举例

我们来看看wxWidgets自带的向导例子.它包含四个页面,如下图所示(页面索引并没有显示在对话框上,这样说只是为了清晰).




Absolutely Useless Wizard
Try checking the box below and
then going back and clearing it
Skip the next page
2
< Back
Next >
Cancel


Absolutely Useless Wizard
Allow to proceed:
forward
backward
both
neither
3
< Back
Next >
Cancel



第一个页面非常简单,它不需要实现任何派生类,只是简单的创建了一个wxWizardPageSimple类的实例,然后在其中增加了一个静态文本标签,如下所示:

```
#include "wx/wizard.h"
wxWizard *wizard = new wxWizard(this, wxID_ANY,
                  wxT("Absolutely Useless Wizard"),
                  wxBitmap(wiztest_xpm),
                  wxDefaultPosition,
                  wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER);
// 第一页
wxWizardPageSimple *page1 = new wxWizardPageSimple(wizard);
wxStaticText *text = new wxStaticText(page1, wxID_ANY,
    wxT("This wizard doesn't help you\nto do anything at all.\n")
    wxT("\n")
    wxT("The next pages will present you\nwith more useless controls."),
         wxPoint(5,5));
```

第二页则实现了一个wxWizardPage的派生类,重载了其GetPrev和GetNext函数.前者总是返回第一页,而后者则可以根据用户的选择返回下一页或者最后一页.其声明和实现如下所示:

```
// 演示怎样动态改变页面顺序
// 第二页
class wxCheckboxPage : public wxWizardPage
{
public:
    wxCheckboxPage(wxWizard *parent,
                   wxWizardPage *prev,
                   wxWizardPage *next)
        : wxWizardPage(parent)
    {
        m_prev = prev;
        m_next = next;
        wxBoxSizer *mainSizer = new wxBoxSizer(wxVERTICAL);
        mainSizer->Add(
            new wxStaticText(this, wxID_ANY, wxT("Try checking the box below and\n")
                                       wxT("then going back and clearing it")),
            0, // 不需要垂直拉伸
            wxALL,
            5 // 边界宽度
        );
        m_checkbox = new wxCheckBox(this, wxID_ANY,
                          wxT("&Skip the next page"));
        mainSizer->Add(
            m_checkbox,
            0, // 不需要垂直拉伸
            wxALL,
            5 // 边界宽度
        );
        SetSizer(mainSizer);
        mainSizer->Fit(this);
    }
    // 重载wxWizardPage函数
    virtual wxWizardPage *GetPrev() const { return m_prev; }
    virtual wxWizardPage *GetNext() const
    {
        return m_checkbox->GetValue() ? m_next->GetNext() : m_next;
    }
private:
    wxWizardPage *m_prev,
                 *m_next;
    wxCheckBox *m_checkbox;
};
```

第三页实现了一个wxRadioboxPage类,它拦截取消向导和页面改变事件.如果你视图在这个页面取消向导,它会询问你是否真的要取消,如果你选择否,则它将使用事件的Veto函数来取消这个操作.OnWizardPageChanging函数则拦截所有的页面改变事件,并根据当前单选框的选项来确定是否允许页面改变.在实际应用程序中,你可以使用这种技术来确保向导在某一页的时候必须填充某些必须的域,否则不可以前进到下一页或者你可以出于某种原因阻止用户返回以前的页面.代码列举如下:

```
// 我们演示了另外一个稍微复杂一些的例子,通过拦截相应事件阻止用户向前或者向后翻页
// 或者让用户确认取消操作.
// 第三页
class wxRadioboxPage : public wxWizardPageSimple
{
public:
    // 方向枚举值
    enum
    {
        Forward, Backward, Both, Neither
    };
    wxRadioboxPage(wxWizard *parent) : wxWizardPageSimple(parent)
    {
        // 应该和上面的枚举值对应
        static wxString choices[] = { wxT("forward"), wxT("backward"), wxT("both"), wxT
("neither") };

        m_radio = new wxRadioBox(this, wxID_ANY, wxT("Allow to proceed:"),
                                 wxDefaultPosition, wxDefaultSize,
                                 WXSIZEOF(choices), choices,
                                 1, wxRA_SPECIFY_COLS);
        m_radio->SetSelection(Both);
        wxBoxSizer *mainSizer = new wxBoxSizer(wxVERTICAL);
        mainSizer->Add(
            m_radio,
            0, // 不伸缩
            wxALL,
            5 // 边界
        );
        SetSizer(mainSizer);
        mainSizer->Fit(this);
    }
    // 事件处理函数
    void OnWizardCancel(wxWizardEvent& event)
    {
        if ( wxMessageBox(wxT("Do you really want to cancel?"), wxT("Question"),
                          wxICON_QUESTION | wxYES_NO, this) != wxYES )
        {
            // 不确认,取消
            event.Veto();
        }
    }
    void OnWizardPageChanging(wxWizardEvent& event)
    {
        int sel = m_radio->GetSelection();

        if ( sel == Both )
            return;
        if ( event.GetDirection() && sel == Forward )
            return;
        if ( !event.GetDirection() && sel == Backward )
            return;
        wxMessageBox(wxT("You can't go there"), wxT("Not allowed"),
                     wxICON_WARNING | wxOK, this);
        event.Veto();
    }
private:
    wxRadioBox *m_radio;
    DECLARE_EVENT_TABLE()
};
```

第四页也是最后一页,wxValidationPage,演示了重载transferDataFromWindow函数以便对复选框控件进行数据校验的方法.transferDataFromWindow在无论向前或者向后按钮被点击的时候都会被调用,而且如果这个函数返回失败,将会取消向前或者向后操作.和所有的对话框用法一

样,你可以不必重载transferDataFromWindow函数而是给对应的控件设置一个验证器.这个页面还演示了怎样更改作为向导构造函数的一个参数的默认的左图片.下面是相关的代码:

```
// 第四页
class wxValidationPage : public wxWizardPageSimple
{
public:
    wxValidationPage(wxWizard *parent) : wxWizardPageSimple(parent)
    {
        m_bitmap = wxBitmap(wiztest2_xpm);
        m_checkbox = new wxCheckBox(this, wxID_ANY,
                                  wxT("&Check me"));
        wxBoxSizer *mainSizer = new wxBoxSizer(wxVERTICAL);
        mainSizer->Add(
            new wxStaticText(this, wxID_ANY,
                     wxT("You need to check the checkbox\n")
                     wxT("below before going to the next page\n")),
            0,
            wxALL,
            5
        );
        mainSizer->Add(
            m_checkbox,
            0,
            wxALL,
            5
        );
        SetSizer(mainSizer);
        mainSizer->Fit(this);
    }
    virtual bool TransferDataFromWindow()
    {
        if ( !m_checkbox->GetValue() )
        {
            wxMessageBox(wxT("Check the checkbox first!"),
                         wxT("No way"),
                         wxICON_WARNING | wxOK, this);
            return false;
        }
        return true;
    }
private:
    wxCheckBox *m_checkbox;
};
```

下面的代码用于将所有的页面放在一起并且开始执行这个向导:

```
void MyFrame::OnRunWizard(wxCommandEvent& event)
{
    wxWizard *wizard = new wxWizard(this, wxID_ANY,
                    wxT("Absolutely Useless Wizard"),
                    wxBitmap(wiztest_xpm),
                    wxDefaultPosition,
                    wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER);
    // 向导页面既可以是一个预定义对象的实例
    wxWizardPageSimple *page1 = new wxWizardPageSimple(wizard);
    wxStaticText *text = new wxStaticText(page1, wxID_ANY,
         wxT("This wizard doesn't help you\nto do anything at all.\n")
         wxT("\n")
         wxT("The next pages will present you\nwith more useless controls."),
         wxPoint(5,5)
        );

    // ... 也可以是一个派生类的实例
    wxRadioboxPage *page3 = new wxRadioboxPage(wizard);
    wxValidationPage *page4 = new wxValidationPage(wizard);
    // 一种方便的设置页面顺序的方法
    wxWizardPageSimple::Chain(page3, page4);
    // 另外一种设置页面顺序的方法
    wxCheckboxPage *page2 = new wxCheckboxPage(wizard, page1, page3);
    page1->SetNext(page2);
    page3->SetPrev(page2);
    // 允许向导设置自适应的大小.
    wizard->GetPageAreaSizer()->Add(page1);
    if ( wizard->RunWizard(page1) )
    {
        wxMessageBox(wxT("The wizard successfully completed"),
         wxT("That's all"), wxICON_INFORMATION | wxOK);
    }
    wizard->Destroy();
}
```

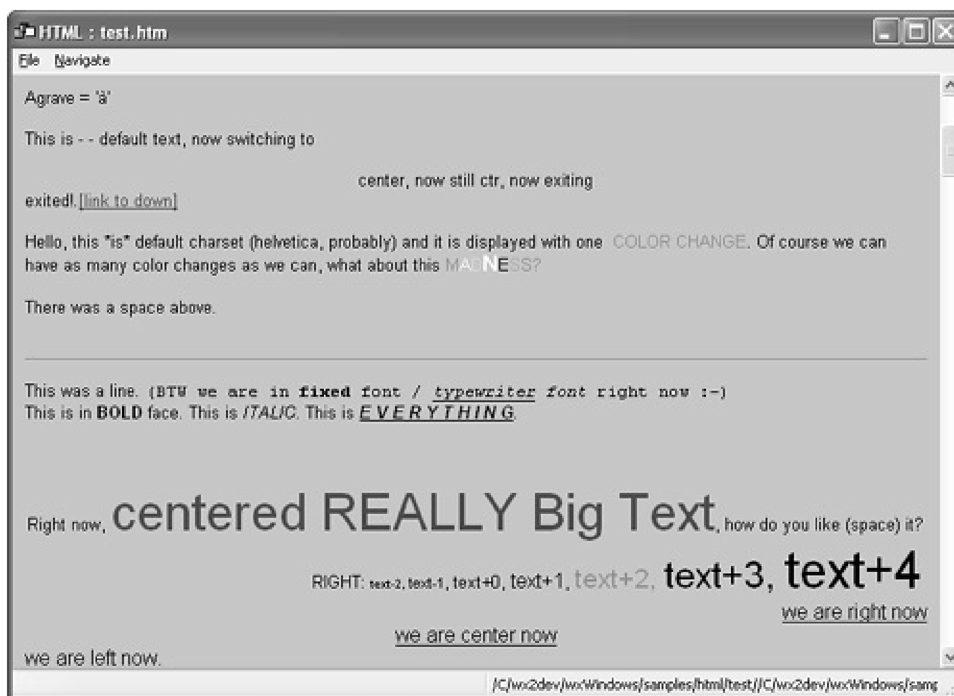
当向导被完成或者取消的时候,MyFrame拦截了相关的事件,在这个例子中,只是简单的将其结果显示在frame窗口的状态条上.当然你也可以在向导类中拦截相应的事件.

完整版本的代码可以在附录J,"代码列表"或者随书光盘的examples/chap12目录中找到.

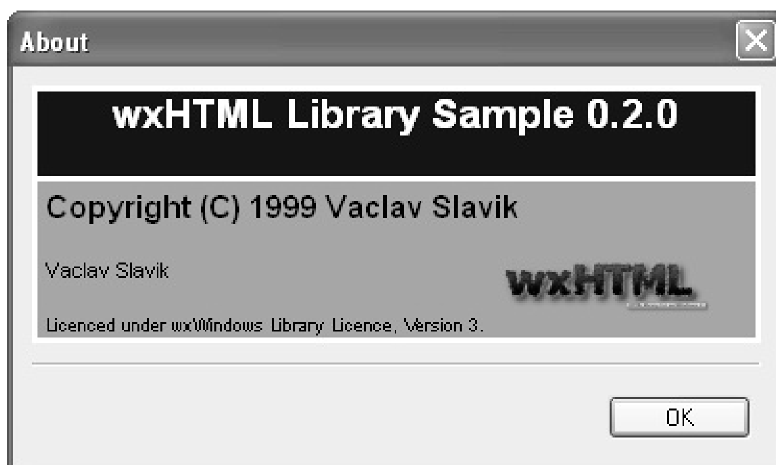
12.4 wxHtmlWindow

wxWidgets在其内建帮助系统中使用了wxHtmlWindow控件,如果你希望你的程序可以显示格式化文件,图片等(比如在生成报表的时候),你可以使用这个控件.这个控件可以支持标准HTML标记的一个子集,包括表格(但是不支持框架),GIF动画,高亮链接显示,字体显示,背景色,列表,居中或者右对齐,水平条以及字符编码等.虽然它不支持CSS,你还是可以通过已经的或者自定义的标记来达到同样的效果.其中的HTML文本也支持拷贝到剪贴板以及从剪贴板以普通文本的方式拷贝回应用程序.

下图演示了自带的samples/html/test例子编译运行以后的样子:

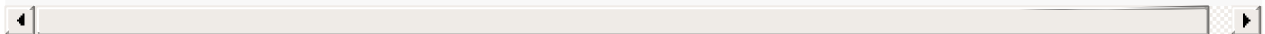


和一个完整的浏览器不同,wxHtmlWindow是小而快的,因此你可以在你的程序中大方的使用它,下图演示了在一个关于对话框中使用wxHtmlWindow的例子:



下面的代码用来创建上面的例子,在这个例子中,HTML控件首先使自己的大小满足其内部的HTML文本的需要,然后对话框的布局控件在调整自己的大小已满足HTML控件的需要.

```
#include "wx/html/htmlwin.h"
void MyFrame::OnAbout(wxCommandEvent& WXUNUSED(event))
{
    wxBoxSizer *topsizer;
    wxHtmlWindow *html;
    wxDialog dlg(this, wxID_ANY, wxString(_("About")));
    topsizer = new wxBoxSizer(wxVERTICAL);
    html = new wxHtmlWindow(&dlg, wxID_ANY, wxDefaultPosition,
            wxSize(380, 160), wxHW_SCROLLBAR_NEVER);
    html->SetBorders(0);
    html->LoadPage(wxT("data/about.htm"));
    // 让HTML控件的大小满足其内部HTML文本需要的大小
    html->SetSize(html->GetInternalRepresentation()->GetWidth(),
                  html->GetInternalRepresentation()->GetHeight());
    topsizer->Add(html, 1, wxALL, 10);
    topsizer->Add(new wxStaticLine(&dlg, wxID_ANY), 0, wxEXPAND | wxLEFT | wxRIGHT, 10);
    wxButton *but = new wxButton(&dlg, wxID_OK, _("OK"));
    but->SetDefault();
    topsizer->Add(but, 0, wxALL | wxALIGN_RIGHT, 15);
    dlg.SetSizer(topsizer);
    topsizer->Fit(&dlg);
    dlg.ShowModal();
}
```



下面列出的是例子中的HTML文本:

```
<html>
<body bgcolor="#FFFFFF">
<table cellspacing=3 cellpadding=4 width="100%">
  <tr>
    <td bgcolor="#101010">
    <center>
    <font size=+2 color="#FFFFFF"><b><br>wxHTML Library Sample 0.2.0<br></b>
    </font>
    </center>
    </td>
  </tr>
  <tr>
    <td bgcolor="#73A183">
    <b><font size=+1>Copyright (C) 1999 Vaclav Slavik</font></b><p>
    <font size=-1>
      <table cellpadding=0 cellspacing=0 width="100%">
        <tr>
          <td width="65%">
            Vaclav Slavik<p>
          </td>
          <td valign=top>
            <img src="logo.png">
          </td>
        </tr>
      </table>
    <font size=1>
    Licenced under wxWindows Library Licence, Version 3.
    </font>
    </font>
    </td>
  </tr>
</table>
</body>
</html>
```

请参考第4章,"基础窗口类"中,"wxListBox和wxCheckListBox"小节中关于wxHtmlListBox的内容.

wxHtmlWindow窗口类型

wxHW_SCROLLBAR_NEVER	不要显示滚动条.
wxHW_SCROLLBAR_AUTO	只在需要的时候显示滚动条.
wxHW_NO_SELECTION	用户不可以选择其中的文本(默认可以).

wxHtmlWindow成员函数

GetInternalRepresentation函数返回最顶层的wxHtmlContainerCell控件,使用它的GetWidth和GetHeight函数可以得到整个HTML区域的大致大小.

LoadFile加载一个HTML文件然后显示它. LoadPage则可以传递一个URL. 有效的URL包括:

http://www.wxwindows.org/front.htm # 一个URL file:myapp.zip#zip:html/index.htm # 一个zip文件中特定的文件

SetPage则直接传递要显示的HTML字符串.

OnCellClicked函数在有鼠标单击某个HTML元素的时候被调用.它的参数包括wxHtmlCell指针, X和Y座标, 一个wxMouseEvent引用.其默认行为为:如果这个元素是一个超链接,则调用OnLinkClicked函数.

OnLinkClicked函数的参数为wxHtmlLinkInfo类型,它的默认行为是调用LoadPage函数加载当前链接.你可以重载这种行为,比如,在你的关于对话框中,你可以在用户单击某个链接的时候使用默认的浏览器打开你的主页.

其它可以重载的函数包括OnOpeningURL,它在某个URL正被打开的时候调用, OnCellMouseHover函数,当鼠标移过某个HTML元素的时候被调用.

ReadCustomization和WriteCustomization函数则用来保存字体和边界信息,它们的参数包括一个wxConfig*指针以及一个可选的位于配置中的路径.

你可以使用SelectAll,SelectLine和SelectWord函数来进行文本选择,SelectionToText将当前选择区域以纯文本的方式返回,ToText函数则将整页以纯文本的方式返回.

SetBorders函数用来设置HTML周围的边框,SetFonts函数则用来设置字体名称,你还可以给七个预定义的字体大小指定整数类型点单位的具体大小.

AppendToPage函数在当前的HTML文本中添加内容并且刷新窗口.

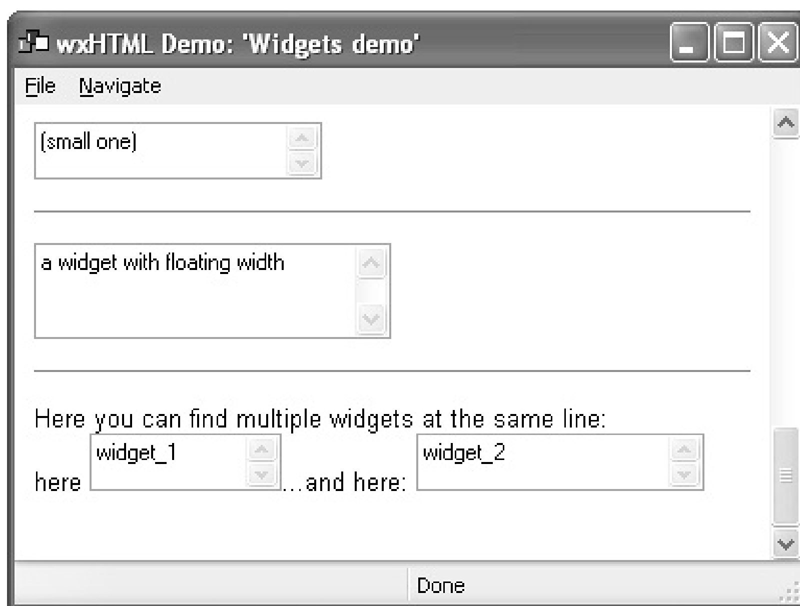
你可以编写一个自定义的wxHtmlFilter以用来读取特定的文件,你可以使用AddFilter函数将其在wxHtmlWindow类中登记.比如,你可以写一个自定义的过滤器用来解密并显示加密的HTML电子杂志.

GetOpenedAnchor, GetOpenedPage和GetOpenedPageTitle函数用来返回当前网页的一些相关信息.

wxHtmlWindow有自己的访问历史机制,你可以通过HistoryBack, HistoryForward, HistoryCanBack, HistoryCanForward和HistoryClear函数来使用它.

在网页中集成窗口控件

你可以在网页中集成你自己的窗口控件,甚至包括那些你自定义的控件,如下图所示.这是通过制作一个定制的标签处理函数来实现的,这个定制的标签处理函数用来处理某些特定的标签并相应的在网页窗口中插入自己的窗口.



上图演示了某个HTML窗口的一部分,是由下面的网页产生的:

```
<mybind name="(small one)" x=150 y=30>
<hr>
<mybind name="a widget with floating width" float=y x="50" y=50>
<hr>
Here you can find multiple widgets at the same line:<br>
 here
<mybind name="widget_1" x="100" y=30>
...and here:
<mybind name="widget_2" x="150" y=30>
```

用于实现特定的HTML标记mybind的代码如下所示:

```
#include "wx/html/m_templ.h"
TAG_HANDLER_BEGIN(MYBIND, "MYBIND")
TAG_HANDLER_PROC(tag)
{
    wxWindow *wnd;
    int ax, ay;
    int fl = 0;
    tag.ScanParam(wxT("X"), wxT("%i"), &ax);
    tag.ScanParam(wxT("Y"), wxT("%i"), &ay);
    if (tag.HasParam(wxT("FLOAT"))) fl = ax;
    wnd = new wxTextCtrl(m_WParser->GetWindow(), wxID_ANY, tag.GetParam(wxT("NAME")),
        wxPoint(0,0), wxSize(ax, ay), wxTE_MULTILINE);
    wnd->Show(true);
    m_WParser->GetContainer()->InsertCell(new wxHtmlWidgetCell(wnd, fl));
    return false;
}
TAG_HANDLER_END(MYBIND)
TAGS_MODULE_BEGIN(MyBind)
    TAGS_MODULE_ADD(MYBIND)
TAGS_MODULE_END(MyBind)
```

这种技术在你想使用wxHtmlWindow来创建整个应用程序界面的时候是比较有用的,你可以用一些教本来产生HTML文件以响应用户输入,就象一个网页表单一样.另外一个例子是当你需要搜集用户的注册信息的时候,你可以提供这样一个界面,其中包含文本框,用户可以输入相关的信息然后点击"注册"按钮以便将他输入的信息发送到你的组织.或者你也可以使用这种技术来给用户产生一个报告,报告中包含一些复选框,以便可以可以针对他感兴趣的内容进行详细查看.

关于制作自定义HTML标记的更详细的内容,请参考samples/html/widget目录中的例子,或者wxWidgets参考手册.

HTML打印

通常如果你的应用程序中使用了wxHtmlWindow,那么你也希望能够打印这些HTML文件.wxWidgets提供了一个 wxHtmlEasyPrinting类来用一种简单的方法打印HTML文件.你需要作的是创建一个这个类的实例,然后使用本地HTML文件调用 PreviewFile和PrintFile函数.你还可以调用PageSetup函数来显示打印设置对话框,使用GetPrintData函数和GetPageSetupData获取用户的打印设置和页面设置.可以通过SetHeader和SetFooter函数定制页眉和页脚,其中可以包含预定义的宏@PAGENUM@(当前页码)和@PAGESCNT@(总页数).

下面的代码取自samples/html/printing,演示了上述控件的基本使用方法以及怎样更改默认的字体:

```
#include "wx/html/htmlwin.h"
#include "wx/html/htmprint.h"
MyFrame::MyFrame(const wxString& title,
                 const wxPoint& pos, const wxSize& size)
        : wxFrame((wxFrame *)NULL, wxID_ANY, title, pos, size)
{
    ...
    m_Name = wxT("testfile.htm");
    m_Prn = new wxHtmlEasyPrinting(_("Easy Printing Demo"), this);
    m_Prn->SetHeader(m_Name + wxT("(@PAGENUM@/@PAGESCNT@)&lt;hr&gt;"),
                     wxPAGE_ALL);
}
MyFrame::~MyFrame()
{
    delete m_Prn;
}
void MyFrame::OnPageSetup(wxCommandEvent& event)
{
    m_Prn->PageSetup();
}
void MyFrame::OnPrint(wxCommandEvent& event)
{
    m_Prn->PrintFile(m_Name);
}
void MyFrame::OnPreview(wxCommandEvent& event)
{
    m_Prn->PreviewFile(m_Name);
}
void MyFrame::OnPrintSmall(wxCommandEvent& event)
{
    int fontsizes[] = { 4, 6, 8, 10, 12, 20, 24 };
    m_Prn->SetFonts(wxEmptyString, wxEmptyString, fontsizes);
}
void MyFrame::OnPrintNormal(wxCommandEvent& event)
{
    m_Prn->SetFonts(wxEmptyString, wxEmptyString, 0);
}
void MyFrame::OnPrintHuge(wxCommandEvent& event)
{
    int fontsizes[] = { 20, 26, 28, 30, 32, 40, 44 };
    m_Prn->SetFonts(wxEmptyString, wxEmptyString, fontsizes);
}

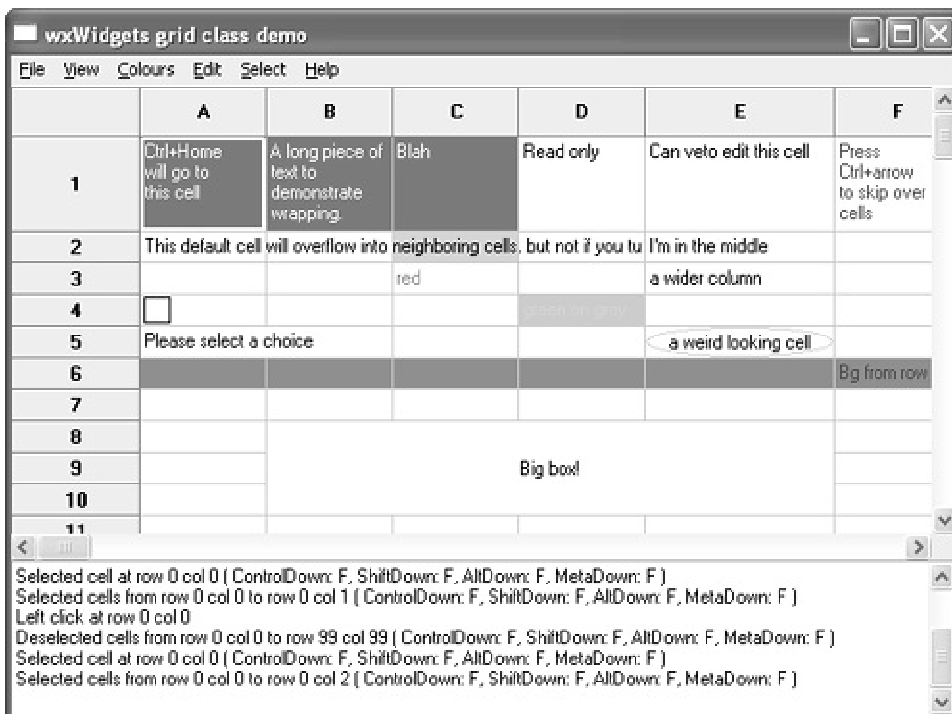
wxWidgets自带的samples/html例子演示了上述所有的知识,请参考.
```

12.5 wxGrid

wxGrid是一个功能强大的但是又稍微有一些复杂的窗口类用来显示表格类型的数据.你还可以使用它来作为一个包含名称和值两栏的属性编辑器.或者是通过你自己的代码使其作为一个一般意义上的表格,用来显示一个数据库或者是你自己应用程序产生的特定统计数据.在某种情况你,你还可以用它来代替列表控件中的报告显示模式,尤其是你希望在某一个特定的表格位置显示图片的时候.

网格控件拥有给电子表格增加行标题或者列标题的能力,用户可以通过拖拽行或者列之间的分割线来改变行列的大小,选择某个或者某几个特定的表格,以及通过点击某一格对其中的值进行编辑.每一个表格都有自己单独的属性包括字体,颜色,对齐方式以及自己的渲染方法(用于绘制表格)和编辑器(用于编辑表格的值).你也可以制作你自己的渲染器和编辑器:参考include/wx/generic/grid.h和 src/generic/grid.cpp中的代码.默认情况下,表格将使用简单文本渲染器和编辑器.如果你使用的表格拥有不同于预定义属性的属性,你可以创建一个"属性提供者"对象,以便在程序运行期动态返回需要的表格的属性.

你也可以使用包含大量数据的虚表格,这种表格的数据由应用程序自己保管,不过不是通过wxGrid类.你需要使用 wxGridTableBase的派生类并且重载其中的GetValue, GetNumberRows和GetNumberCols函数.这些函数将和你的应用程序数据(也许是数据库)打交道.然后通过SetTable函数给网格控件增加一份数据以便网格控件可以进行对应的显示,这种更深入的技巧在你的wxWidgets发行版的samples/grid目录中进行了演示,如下图所示:



下面的代码创建了一个简单的8行10列的表格:

```
#include "wx/grid.h"
// 创建一个wxGrid对象
wxGrid* grid = new wxGrid(frame, wxID_ANY,
                          wxDefaultPosition, wxSize(400, 300));
// 然后调用CreateGrid设置表格的大小
grid->CreateGrid(8, 10);
// 我们可以单独给某一行或者某一列设置象素单位的宽度或高度.
grid->SetRowSize(0, 60);
grid->SetColSize(0, 120 );
// 然后设置表格内的文本内容
grid->SetCellValue(0, 0, wxT("wxGrid is good"));
// 可以指定某些表格是只读的
grid->SetCellValue(0, 3, wxT("This is read-only"));
grid->SetReadOnly(0, 3);
// 还可以指定某个表格的文本颜色
grid->SetCellValue(3, 3, wxT("green on grey"));
grid->SetCellTextColour(3, 3, *wxGREEN);
grid->SetCellBackgroundColour(3, 3, *wxLIGHT_GREY);
// 还可以指定某一列的数据采用数字的格式
// 这里我们设置第5列的数字格式为宽度为6,小数点后保留2位的浮点数.
grid->SetColFormatFloat(5, 6, 2);
grid->SetCellValue(0, 6, wxT("3.1415"));
// 设置网格的大小为可以显示所有内容的最小大小.
grid->Fit();
// 设置其父窗口的客户区大小以便放的下整个网格.
frame->SetClientSize(grid->GetSize());
```

wxGrid系统中的类

正如你已经理解到的那样,wxGrid类是许多的类交互作用的结果.下表展示了wxGrid系统中的类以及它们之间的相互关系:

wxGrid	最主要的网格类,用来存放别的用于管理表格,行和列等的其它的窗口类.
wxGridTableBase	这个类允许应用程序想虚拟的网格提供数据.SetTable函数将其派生类的一个实例挂载入网格类.
wxGridCellAttr	保存用于渲染表格的属性数据.你可以显式的通过类似SetCellTextColour这样的函数更改表格的属性.你也可以通过SetAttr函数设置某个单独的表格的属性或者是通过SetRowAttr和SetColAttr函数设置某一列或者某一行的属性. 你也可以在你自定义的表格类中通过GetAttr函数返回指定表格的属性.
wxGridCellRenderer	这个类负责对表格进行渲染和绘制.你可以通过改变wxGridCellAttr(或者通过wxGrid::SetCellRenderer函数)中对应的类的实例来改变某个表格的数据格式,你也可以直接通过wxGrid::SetDefaultRenderer函数更改整个表格的显示方式.这是一个虚类, 你通常需要使用一个预定义的派生类或者自己实现一个派生类.预定义的派生类包括wxGridCellStringRenderer, wxGridCellNumberRenderer,wxGridCellFloatRenderer和wxGridCellBoolRenderer 等.
wxGridCellEditor	这个类负责实现对表格数据的即时编辑功能.这个虚类的派生类的实例可以和某个表格,某行,某列甚至整个表格绑定. 比如说,使用wxGrid::SetCellEditor函数来给某个表格中的一格设置编辑器. 预定义的派生类包括wxGridCellTextEditor, wxGridCellFloatEditor, wxGridCellBoolEditor, wxGridCellNumberEditor和wxGridCellChoiceEditor.
wxGridEvent	这个类包含了各种网格相关事件的信息,比如鼠标在表格上单击事件,表格数据改变事件,表格被选中事件,表格编辑器被显示或者隐藏事件等.
wxGridRangeSelectEvent	当用户选择一组表格以后将产生这个事件.
wxGridSizeEvent	当某一行或者某一列的大小发生变化的时候产生这个事件.
wxGridEditorCreatedEvent	当创建某个编辑器的时候产生这个事件.
wxGridCellCoords	这个类用来代表表格中的某一格. 使用GetRow和GetCol函数获取具体的位置.
wxGridCellCoordsArray	这是一个wxGridCellCoords类型的数组,用在函数GetSelectedCell, GetSelectionBlockTopLeft和GetSelectionBlockBottomRight的返回值中.

wxGrid的事件

下面列出了wxGrid的主要的事件映射宏.注意对于其中任何一个EVT*GRID*...宏,都对应的还有一个EVT*GRID_CMD*...宏,后者比前者多一个标识符参数,可以用来避免仅仅出于这个原因而重新定义新的类.

EVT_GRID_CELL_LEFT_CLICK(func)	用户用左键单击某个表格.
EVT_GRID_CELL_RIGHT_CLICK(func)	用户用邮件单击某个表格.
EVT_GRID_CELL_LEFT_DCLICK(func)	用户用左键双击某个表格.
EVT_GRID_CELL_RIGHT_DCLICK(func)	用户用右键双击某个表格.
EVT_GRID_LABEL_LEFT_CLICK(func)	用户用左键单击某个标题.
EVT_GRID_LABEL_RIGHT_CLICK(func)	用户用右键单击某个标题.
EVT_GRID_LABEL_LEFT_DCLICK(func)	用户用左键双击某个标题.
EVT_GRID_LABEL_RIGHT_DCLICK(func)	用户用右键双击某个标题.
EVT_GRID_CELL_CHANGE(func)	用户更改了某个表格的数据.
EVT_GRID_SELECT_CELL(func)	用户选中了某个表格.
EVT_GRID_EDITOR_HIDDEN(func)	某个表格的编辑器已隐藏.
EVT_GRID_EDITOR_SHOWN(func)	某个表格的编辑器已显示.
EVT_GRID_COL_SIZE(func)	用户通过拖拽改变了某列大小.
EVT_GRID_ROW_SIZE(func)	用户通过拖拽改变某行大小.
EVT_GRID_RANGE_SELECT(func)	用户选取了一组连续的单元格.
EVT_GRID_EDITOR_CREATED(func)	某个单元格的编辑器已被创建.

wxGrid的成员函数

下面按功能列出了比较重要的wxGrid的成员函数.完整的成员函数列表以及其它相关类的成员函数请参考相关手册.

用于创建,删除和数据交互的函数

AppendCols和AppendRows用来在最下边或者最右边增加行或者列,或者你还可以使用InsertCols和InsertRows在某个特定位置插入行或者列. 如果你使用了自定义的表格,你需要重载同名的这些函数.

GetNumberCols和GetNumberRows函数用来获取和网格绑定的表格数据的列数和行数.

CreateGrid这个函数用来以指定的行数和列数初始化网格的数据.你应该在创建网格实例以后马上调用这个函数.这个函数会为网格创建一个用于操作简单文本数据的表格,表格的所有相关数据都将保存在内存中.如果你的应用程序要处理更负责的数据类型或者更复杂的依赖关系,或者说要处理很大量的数据,你可以实现自己的表格派生类,然后使用wxGrid::SetTable函数将其和某个网格对象绑定.

ClearGrid函数清除所有的网格绑定表格中的数据并且刷新网格的显示.表格本身并不会被释放.如果你使用了自定义的表格类,记得重载其wxGridTableBase::Clear成员函数以实现对应功能. ClearSelection函数用来去选择所有当前选择的单元格.

DeleteCols和DeleteRows函数分别用来删除列和行.

GetColLabelValue函数返回某个指定列的标签.默认的表格类提供的列标签是从A, B�Z, AA, AB�ZZ, AAA�等,如果你使用自定义的表格类,你可以重定义 wxGridTableBase::GetColLabelValue函数以返回相应的标签. 类似的, GetrowLabelValue函数用来返回指定行的标签. 默认的行标签为数字.如果你使用自定义的表格,可以重载其 wxGridTableBase::GetRowLabelValue函数来提供自定义的默认行标签.你还可以使用 SetColLabelValue和SetRowLabelValue函数来指定某个特定行列的标签.

GetCellValue用来返回特定单元格内的文本.对于那些使用简单网格类的应用程序,你可以使用这个函数和对应的SetCellValue函数来操作单元格内的文本数据.对于更复杂的使用了自定义表格的程序,这个函数只能用于返回那些包含文本内容的单元格的内容.

界面相关函数

一下这些函数将会影响到网格控件的界面更新

BeginBatch和EndBatch函数阻止它们之间的对网格对象的操作引起的界面更新. 界面更新将在GetBatchCount返回0的情况下更新(译者注:BeginBatch增加这个值而EndBatch减少这个值).

EnableGridLines设置允许或者禁止绘制网格线. GridLinesEnabled则返回当前设置.

ForceRefresh用来强制立即更新网格显示. 你应该使用这个函数来代替wxWindow::Refresh函数.

Fit函数用来使得网格控件将自己的大小更改为当前行数和列数所要求的最小大小.

GetCellAlignment返回指定单元格在垂直和水平方向上的对齐方式. GetColLabel返回列标签的对齐方式, GetRowLabelAlignment返回行标签的对齐方式. GetDefaultCellAlignment返回默认单元格的对齐方式. 这些函数都有对应的Set开始的设置函数. 水平对齐的值可以为 wxALIGN_LEFT, wxALIGN_CENTRE (wxALIGN_CENTER)或wxALIGN_RIGHT. 垂直对齐的枚举值可以为wxALIGN_TOP, wxALIGN_CENTRE (wxALIGN_CENTER)或 wxALIGN_BOTTOM之一.

GetCellBackgroundColour返回指定单元格背景颜色. GetdefaultCellBackgroundColour返回默认单元格背景颜色.GetLabelBackgroundColour返回标签背景颜色.这些函数也都有对应的设置函数.

GetCellFont函数返回指定单元格字体, GetdefaultCellFont返回默认单元格字体. GetLabelFont则返回标签文本字体.这些函数也都有对应的设置函数.

GetCellTextColour返回指定单元格的文本颜色. GetdefaultCellTextColour返回默认单元格的文本颜色,GetLabelTextColour则返回标签文本颜色.这些函数同样拥有对应的设置函数

SetGridLineColour和GetGridLineColour函数用来更改和获取网格线的颜色.

SetColAttr和SetRowAttr用来设置某一行或者某一列中所有单元格的属性.

SetColFormatBool, SetColFormatNumber, SetColFormatFloat和SetColFormatCustom函数可以用来更改某一列的单元格内容格式.

和wxGrid大小相关的函数

下面的这些函数的参数都以象素作为单位.

AutoSize函数自动将所有单元格的大小按照其内容进行调整.对应的还有AutoSizeColumn, AutoSizeColumns, AutoSizeRow和AutoSizeRows函数.

CellToRect返回指定单元格的大小和位置,以wxRect表示.

SetColMinimalWidth和SetRowMinimalHeight用来设置最小行高和最小列宽, 对应的获取函数为GetColMinimalWidth和GetrowMinimalHeight.

下面这些函数用来获取各种尺寸大小: GetColLabelSize, GeTDefaultColLabelSize, GeTDefaultColSize, GetColSize, GetdefaultRowLabelSize, GeTRowSize, GeTDefaultRowSize和GeTRowLabelSize. 它们都有对应的设置函数.

如果你想在表格周围设置额外的边距,可以使用SetMargins函数.

XToCol和YToRow函数用来将X和Y座标转换成对应的行数或列数.要找到距离其右边线最近的对应于X座标值的列号,使用XToEdgeOfCol函数. 要找到具体其底边线最近的对应于Y座标的行号,使用YToEdgeOfRow函数.

选择和游标函数

下面的这些函数用于控制网格的游标和选择操作

GetGridCursorCol和GetGridCursorRow函数返回当前游标所处的行号和列号.相应的设置函数为SetGridCursor.

你可以用下面的函数以每次一格的方式移动游标:MoveCursorDown, MoveCursorLeft, MoveCursorRight和MoveCursorUp.如果希望在移动的时候跳到第一个非空单元格,则对应的使用 MoveCursorDownBlock, MoveCursorLeftBlock, MoveCursorRightBlock和MoveCursorUpBlock函数.

如果想一次移动一页,可以使用MovePageDown和MovePageUp函数,页大小由网格窗口的大小决定.

GetSelectionMode返回当前设置的选择模式,它的值为下列之一: wxGrid::wxGridSelectCells (默认模式,以单元格为单位选择), wxGrid::wxGridSelectRows (以行为单位选择), and wxGrid::wxGridSelectColumns (以列为单位进行选择). 对应的设置函数为SetSelectionMode.

GetSelectedCells用来获取当前选中的单元格列表,它的返回值是wxGridCellCoordsArray类型,其中包含所有被选中的单元格.GetSelectedCols和GetSelectedRows则返回当前选中的所有行和列.因为用户可以选择多个不连续的单元格块,所以GetSelectionBlockTopLeft和GetSelectionBlockBottomRight返回的也是一个 wxGridCellCoordsArray类型的类表. 你需要自己群举这些列表以遍历所有选中的单元格.

IsInSelection用于查询指定的单元格是否被选中,参数为行号列号或者wxGridCellCoords类型. IsSelection则返回整个网格是否有任何一格单元格被选中.

使用SelectAll选择整个表格, SelectCol函数用于选择某列, SelectRow函数用于选择某行. SelectBlock用于选择连续的一块区域,参数为四个代表左上角及右下角的座标的整数,或者两个wxGridCellCoords类型的参数.

其它wxGrid函数

GetTable函数返回网格用于保存内部数据的绑定表格对象.如果你使用了CreateGrid函数,则已经创建了一个用户保存简单文本数据的表格,或者,你已经使用SetTable函数设置了一个你自定义的表格类型.

GetCellEditor和SetCellEditor用来获取或者设置指定单元格绑定的编辑器指针. GetdefaultEditor和SetDefaultEditor用来获取和设置所有单元格使用的默认编辑器.

GetCellRenderer和SetCellRenderer获取或者设置指定单元格绑定的渲染器的指针. GeTDefaultRenderer和SetDefaultRenderer用来获取和设置所有单元格使用的渲染器.

ShowCellEditControl和HideCellEditControl用来在相应的位置显示和隐藏当前单元格指定的编辑控件.这两个函数通常是在用户点击某个单元格来编辑单元格的值或者按下回车键和取消键(或者单击另外一个窗口)以关闭编辑器的时候自动调用的. SaveEditControlValue函数用来将编辑器中的值传输到单元格中,这个函数在关闭网格或者从网格获取数据的时候你可以考虑调用以便网格的值反应最新编辑的值.

EnableCellEditControl函数允许或者禁止对网格数据进行编辑. 这个函数调用的时候,网格类将会产生wxEVT_GRID_EDITOR_SHOWN或wxEVT_GRID_EDITOR_HIDDEN事件. IsCellEditControlEnabled函数用来检测是否当前单元格可以被编辑. IsCurrentCellReadOnly则返回是否当前单元格是只读的.

EnableDragColSize允许或者禁止通过拖拽更改列宽. EnableDragGridSize允许或者禁止通过拖拽改变单元格大小. EnableDragRowSize允许或者禁止通过拖拽改变行高.

EnableEditing的参数如果为false,则设置整个网格为只读状态. 如果为true,则网格恢复到默认状态.在默认状态,单元格和行以及列都可以有自己的是否可编辑标记,这个标记可以通过wxGridCellAttribute::SetReadOnly改变,针对单元格的只读设置可以直接通过wxGridCellAttribute:: SetReadOnly函数改变.IsEditable函数返回是否整个网格是可编辑的.

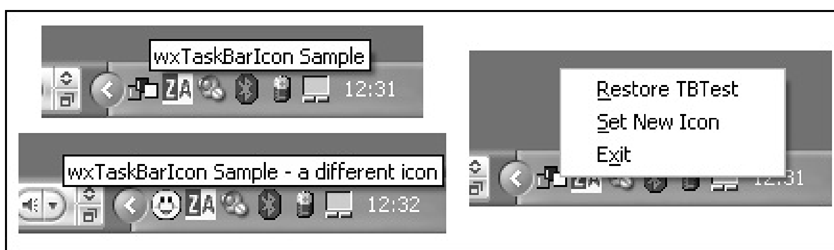
你也可以通过调用SetReadOnly函数改变某个单元格的只读状态,IsReadOnly则用来获取这个设置.

IsVisible函数在单元格部分或者全部可见的时候返回true,MakeCellVisible则确保某个单元格出于可见区域.

12.6 wxTaskBarIcon

这个类的功能是在系统托盘区(Windows,Gnome或者KDE)或者停靠区(Mac OS X)安装一个图标.点击这个图标将会弹出一个应用程序提供的菜单,并且在当鼠标划过图标的时候会显示一个可选的工具提示.这种技术提供了一种不必通过正规的用户界面快速访问某些重要功能的方法.应用程序可以通过更换图标来提供某些状态信息,比如用来提示电池的剩余电量或者windows系统上的网络连接提示.

下图演示了wxWidgets自带的samples/taskbar例子在windows平台上运行时的样子.它首先显示一个 wxWidgets的图标,当鼠标移过这个图标的时候显示"wxTaskBarIconSample."工具提示,右键单击这个图标将会显示一个菜单,菜单有三个选项,选择设置新图标选项将会将图标设置为一个笑脸,并且把工具提示更改为一个新的文本.



以wxTaskBarIcon方式驱动的应用程序并不十分复杂,如下所示.自定义的类型MyTaskBarIcon重载了wxTaskBarIcon的CreatePopupMenu函数,并且拦截了左键双击事件,并实现了三个菜单项.

```
class MyTaskBarIcon: public wxTaskBarIcon
{
public:
    MyTaskBarIcon() {};
    void OnLeftButtonDClick(wxTaskBarIconEvent&);
    void OnMenuRestore(wxCommandEvent&);
    void OnMenuExit(wxCommandEvent&);
    void OnMenuSetNewIcon(wxCommandEvent&);
    virtual wxMenu *CreatePopupMenu();
DECLARE_EVENT_TABLE()
};
enum {
    PU_RESTORE = 10001,
    PU_NEW_ICON,
    PU_EXIT,
};
BEGIN_EVENT_TABLE(MyTaskBarIcon, wxTaskBarIcon)
    EVT_MENU(PU_RESTORE, MyTaskBarIcon::OnMenuRestore)
    EVT_MENU(PU_EXIT,    MyTaskBarIcon::OnMenuExit)
    EVT_MENU(PU_NEW_ICON,MyTaskBarIcon::OnMenuSetNewIcon)
    EVT_TASKBAR_LEFT_DCLICK  (MyTaskBarIcon::OnLeftButtonDClick)
END_EVENT_TABLE()
void MyTaskBarIcon::OnMenuRestore(wxCommandEvent& )
{
    dialog->Show(true);
}
void MyTaskBarIcon::OnMenuExit(wxCommandEvent& )
{
    dialog->Close(true);
}
void MyTaskBarIcon::OnMenuSetNewIcon(wxCommandEvent&)
{
    wxIcon icon(smile_xpm);

    if (!SetIcon(icon, wxT("wxTaskBarIcon Sample - a different icon")))
        wxMessageBox(wxT("Could not set new icon."));
}
// 重载
wxMenu *MyTaskBarIcon::CreatePopupMenu()
{
    wxMenu *menu = new wxMenu;
    menu->Append(PU_RESTORE, wxT("&Restore TBTest"));
    menu->Append(PU_NEW_ICON,wxT("&Set New Icon"));
    menu->Append(PU_EXIT,    wxT("E&xit"));
    return menu;
}
void MyTaskBarIcon::OnLeftButtonDClick(wxTaskBarIconEvent&)
{
    dialog->Show(true);
}
```

下面的代码则用来显示一个对话框并且安装初始化图标.

```
#include "wx/wx.h"
#include "wx/taskbar.h"
// 定义一个新的应用程序
class MyApp: public wxApp
{
public:
    bool OnInit(void);
};
class MyDialog: public wxDialog
{
public:
    MyDialog(wxWindow* parent, const wxWindowID id, const wxString& title,
        const wxPoint& pos, const wxSize& size, const long windowStyle =
 wxDEFAULT_DIALOG_STYLE);
    ~MyDialog();
```

```
    void OnOK(wxCommandEvent& event);
    void OnExit(wxCommandEvent& event);
    void OnCloseWindow(wxCloseEvent& event);
    void Init(void);
protected:
    MyTaskBarIcon   *m_taskBarIcon;
DECLARE_EVENT_TABLE()
};
#include "../sample.xpm"
#include "smile.xpm"
MyDialog   *dialog = NULL;
IMPLEMENT_APP(MyApp)
bool MyApp::OnInit(void)
{
    // 创建主窗口
    dialog = new MyDialog(NULL, wxID_ANY, wxT("wxTaskBarIcon Test Dialog"),
 wxDefaultPosition, wxSize(365, 290));
    dialog->Show(true);
    return true;
}
BEGIN_EVENT_TABLE(MyDialog, wxDialog)
    EVT_BUTTON(wxID_OK, MyDialog::OnOK)
    EVT_BUTTON(wxID_EXIT, MyDialog::OnExit)
    EVT_CLOSE(MyDialog::OnCloseWindow)
END_EVENT_TABLE()
MyDialog::MyDialog(wxWindow* parent, const wxWindowID id, const wxString& title,
    const wxPoint& pos, const wxSize& size, const long windowStyle):
  wxDialog(parent, id, title, pos, size, windowStyle)
{
    Init();
}
MyDialog::~MyDialog()
{
    delete m_taskBarIcon;
}
void MyDialog::OnOK(wxCommandEvent& WXUNUSED(event))
{
    Show(false);
}
void MyDialog::OnExit(wxCommandEvent& WXUNUSED(event))
{
    Close(true);
}
void MyDialog::OnCloseWindow(wxCloseEvent& WXUNUSED(event))
{
    Destroy();
}
void MyDialog::Init(void)
{
  (void)new wxStaticText(this, wxID_ANY, wxT("Press 'Hide me' to hide me, Exit to quit.")
                         wxPoint(10, 20));
  (void)new wxStaticText(this, wxID_ANY, wxT("Double-click on the taskbar icon to show me
 again."),
                         wxPoint(10, 40));
  (void)new wxButton(this, wxID_EXIT, wxT("Exit"), wxPoint(185, 230), wxSize(80, 25));
  (new wxButton(this, wxID_OK, wxT("Hide me"), wxPoint(100, 230), wxSize(80,
 25)))->SetDefault();
  Centre(wxBOTH);
  m_taskBarIcon = new MyTaskBarIcon();
  if (!m_taskBarIcon->SetIcon(wxIcon(sample_xpm), wxT("wxTaskBarIcon Sample")))
        wxMessageBox(wxT("Could not set icon."));
}
```

wxTaskBarIcon的事件

下表列出的事件宏用来拦截wxTaskBarIcon相关的事件.注意不是所有的发行版都产生这些事件,因此如果你想再鼠标点击的时候弹出菜单,你应该使用重载CreatePopupMenu函数的方法.还要注意wxTaskBarIconEvent事件不会提供任何鼠标指针状态信息,比如鼠标位置之类.

EVT_TASKBAR_MOVE(func)	鼠标正在图标上移动.
EVT_TASKBAR_LEFT_DOWN(func)	左键按下.
EVT_TASKBAR_LEFT_UP(func)	左键释放.
EVT_TASKBAR_RIGHT_DOWN(func)	右键按下.
EVT_TASKBAR_RIGHT_UP(func)	右键释放.
EVT_TASKBAR_LEFT_DCLICK(func)	左键双击.
EVT_TASKBAR_RIGHT_DCLICK(func)	右键双击.

wxTaskBarIcon成员函数

wxTaskBarIcon的成员函数是非常简单的,下面列出的就是它所有的成员函数.

CreatePopupMenu是一个虚函数,应用程序已经重载这个函数以返回一个wxMenu指针.这个函数在对应的 wxEVT_TASKBAR_RIGHT_DOWN事件中被调用(在Mac OS X系统上模拟了这个事件). wxWidgets也会在菜单被关闭的时候自动释放这个菜单所占用的内存.

IsIconInstalled返回是否SetIcon已经被成功调用.

IsOk在wxTaskBarIcon对象已经被成功初始化的时候返回True.

PopupMenu在当前位置显示一个菜单.最好不要调用这个函数,而应该重载CreatePopupMenu函数,然后让wxWidgets帮你显示对应的菜单.

RemoveIcon移除前一次使用SetIcon函数设置的图标.

SetIcon设置一个图标(wxIcon)以及一个可选的工具提示.这个函数可以被多次调用.

12.7 编写自定义的控件

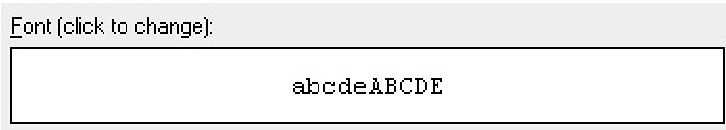
这一小节,我们来介绍一下怎样创建自定义的控件.实际上,wxWidgets并不具有象别的应用程序开发平台上的二进制的,支持鼠标拖入应用程序窗口的这种控件.第三方控件通常都和wxWidgets自带的控件比如wxCalendarCtrl和wxGrid一样,是通过源代码的方式提供的.我们这里用的 "控件"一词,含义是比较松散的,你不一定非要从"wxControl"进行派生,比如有时候,你可能更愿意从wxScrolledWindow产生派生类.

制作一个自定义的控件通常需要经过下面10个步骤:

1. 编写类声明,它应该拥有一个默认构造函数,一个完整构造函数,一个Create函数用于两步创建, 最好还有一个Init函数用于初始化内部数据.
2. 增加一个函数DoGetBestSize,这个函数应该根据内部控件的情况(比如标签尺寸)返回该控件最合适的大小.
3. 如果已有的事件类不能满足需要,为你的控件增加新的事件类. 比如对于内部的一个按钮被按下的事件,可能使用已有的wxCommandEvent就可以了,但是更复杂的控件需要更复杂的事件类.并且如果你增加了新的事件类,也应改增加相应的事件映射宏.
4. 编写代码在你的新控件上显示信息.
5. 编写底层鼠标和键盘控制代码,并在其处理函数中产生你自定义的新的事件,以便应用程序可以作出相应处理.
6. 编写默认事件处理函数,以便控件可以处理那些标准事件(比如处理wxID_COPY或wxID_UNDO等标准命令的事件)或者默认的用户界面更新事件.
7. 可选的增加一个验证器类,以便应用程序可以用它使得数据和控件之间的传输变得容易,并且增加数据校验功能.
8. 可选的增加一个资源处理类,以便可以在XRC文件中使用你自定义的控件.
9. 在你准备使用的所有平台上测试你的自定义控件.
10. 编写文档

我们来举一个简单的例子,这个例子我们在第三章"事件处理"中曾经使用过,当时我们用来讨论自定义的事件: wxFontSelectorCtrl,你可以在随书光盘的examples/chap03中找到这个例子.这个类提供了一个字体预览窗口,当用户在这个窗口上点击鼠标的时候会弹出一个标准的字体选择窗口用于更改当前字体.这将导致一个新的事件wxFontSelectorCtrlEvent,应用程序可以使用EVT_FONT_SELECTION_CHANGED(id, func)宏来拦截这个事件.

这个控件的运行效果如下图所示,下图中静态文本下方即为我们自定义的控件.



自定义控件的类声明

下面的代码展示了自定义控件wxFontSelectorCtrl的类声明.其中DoGetBestSize只简单的返回固定值200x40象素,如果构造函数中没有指定大小,我们会默认使用这个大小.

```
/*!
 * 一个显示显示字体预览的自定义控件.
 */
class wxFontSelectorCtrl: public wxControl
{
    DECLARE_DYNAMIC_CLASS(wxFontSelectorCtrl)
    DECLARE_EVENT_TABLE()
public:
    // 默认构造函数
    wxFontSelectorCtrl() { Init(); }
    wxFontSelectorCtrl(wxWindow* parent, wxWindowID id,
        const wxPoint& pos = wxDefaultPosition,
        const wxSize& size = wxDefaultSize,
        long style = wxSUNKEN_BORDER,
        const wxValidator& validator = wxDefaultValidator)
    {
        Init();
        Create(parent, id, pos, size, style, validator);
    }
    // Create函数
    bool Create(wxWindow* parent, wxWindowID id,
        const wxPoint& pos = wxDefaultPosition,
        const wxSize& size = wxDefaultSize,
        long style = wxSUNKEN_BORDER,
        const wxValidator& validator = wxDefaultValidator);
    // 通用初始化函数
    void Init() { m_sampleText = wxT("abcdeABCDE"); }
    // 重载函数
    wxSize DoGetBestSize() const { return wxSize(200, 40); }
    // 事件处理函数
    void OnPaint(wxPaintEvent& event);
    void OnMouseEvent(wxMouseEvent& event);
    // 操作函数
    void SetFontData(const wxFontData& fontData) { m_fontData = fontData; };
    const wxFontData& GetFontData() const { return m_fontData; };
    wxFontData& GetFontData() { return m_fontData; };
    void SetSampleText(const wxString& sample);
    const wxString& GetSampleText() const { return m_sampleText; };
protected:
    wxFontData  m_fontData;
    wxString    m_sampleText;
};
```

象wxFontDialog中的那样,我们使用wxFontData类型来保存字体数据,这个类型可以额外的保存字体颜色信息.

这个控件的RTTI(运行期类型标识)事件表和创建函数的代码列举如下:

```
BEGIN_EVENT_TABLE(wxFontSelectorCtrl, wxControl)
    EVT_PAINT(wxFontSelectorCtrl::OnPaint)
    EVT_MOUSE_EVENTS(wxFontSelectorCtrl::OnMouseEvent)
END_EVENT_TABLE()
IMPLEMENT_DYNAMIC_CLASS(wxFontSelectorCtrl, wxControl)
bool wxFontSelectorCtrl::Create(wxWindow* parent, wxWindowID id,
             const wxPoint& pos, const wxSize& size, long style,
             const wxValidator& validator)
{
    if (!wxControl::Create(parent, id, pos, size, style, validator))
        return false;
    SetBackgroundColour(wxSystemSettings::GetColour(
                                        wxSYS_COLOUR_WINDOW));
    m_fontData.SetInitialFont(GetFont());
    m_fontData.SetChosenFont(GetFont());
    m_fontData.SetColour(GetForegroundColour());
    // 这个函数告诉相应的布局控件使用指定的最佳大小.
    SetBestFittingSize(size);
    return true;
}
```

对于函数SetBestFittingSize的调用告诉布局控件或者使用构造函数中指定的大小,或者使用DoGetBestSize函数返回的大小作为这个控件的最小尺寸.当控件被增加到一个布局控件中时,根据增加函数的参数不同,这个控件的尺寸有可能被放大.

增加DoGetBestSize函数

实现DoGetBestSize函数的目的是为了让wxWidgets可以以此作为控件的最小尺寸以便更好的布局.如果你提供了这个函数,用户就可以在创建控件的时候使用默认的大小(wxDefaultSize)以便控件自己决定自己的大小.在这里我们只是选择一个固定值200x40象素,虽然是固定的,但是是合理的.当然,应用程序可以通过在构造函数中传递不同的大小来覆盖它.类似按钮或者标签这样的控件,我们可以提供一个合理的大小,当然,你的控件也可能不能够决定自己的大小,比如一个没有子窗口的滚动窗口通常无法知道自己最合适的大小,在这种情况下,你可以不理会wxWindow::DoGetBestSize.在这种情况下,你的控件大小将取决于用户在构造函数中指定的非默认大小或者应用程序需要通过一个布局控件来自动觉得你的控件的大小.

如果你的控件包含拥有可感知大小的子窗口,你可以通过所有子窗口的大小来决定你自己控件的大小,子窗口的合适大小可以通过 GetAdjustedBestSize函数获得.比如如果你的控件包含水平平铺的两个子窗口,你可以用下面的代码来实现DoGetBestSize函数:

```
wxSize ContainerCtrl::DoGetBestSize() const
{
    // 获取子窗口的最佳大小
    wxSize size1, size2;
    if ( m_windowOne )
        size1 = m_windowOne->GetAdjustedBestSize();
    if ( m_windowTwo )
        size2 = m_windowTwo->GetAdjustedBestSize();
    // 因为子窗口是水平平铺的,因此
    // 通过下面的方法计算控件的最佳大小.
    wxSize bestSize;
    bestSize.x = size1.x + size2.x;
    bestSize.y = wxMax(size1.y, size2.y);
    return bestSize;
}
```

定义一个新的事件类

我们在第三章中详细介绍了怎样创建一个新的事件类(wxFontSelectorCtrlEvent)及其事件映射宏 (EVT_FONT_SELECTION_CHANGED).使用这个控件的应用程序可能并不需要拦截这个事件,因为可以直接使用数据传送机制会更方便. 不过在一个更复杂的例子中,事件类可以提供特别的函数,以便应用程序的事件处理函数可以从事件中获取更有用的信息,比如在这个例子中,我们可以增加 wxFontSelectorCtrlEvent::GetFont函数以返回用户当前选择的字体.

显示控件信息

我们的自定义控件使用一个简单的重绘函数显示控件信息(一个居中放置的使用指定字体的文本),如下所示:

```
void wxFontSelectorCtrl::OnPaint(wxPaintEvent& event)
{
    wxPaintDC dc(this);
    wxRect rect = GetClientRect();
    int topMargin = 2;
    int leftMargin = 2;
    dc.SetFont(m_fontData.GetChosenFont());
    wxCoord width, height;
    dc.GetTextExtent(m_sampleText, & width, & height);
    int x = wxMax(leftMargin, ((rect.GetWidth() - width) / 2)) ;
    int y = wxMax(topMargin, ((rect.GetHeight() - height) / 2)) ;
    dc.SetBackgroundMode(wxTRANSPARENT);
    dc.SetTextForeground(m_fontData.GetColour());
    dc.DrawText(m_sampleText, x, y);
    dc.SetFont(wxNullFont);
}
```

如果你需要绘制标准元素,比如分割窗口的分割条或者一个边框,你可以考虑使用wxNativeRenderer类(更多信息请参考使用手册).

处理输入

我们的控件会拦截左键单击事件来显示一个字体对话框,如果用户选择了新的字体,则一个新的事件会使用ProcessEvent函数发送给这个控件.这个事件可以被wxFontSelectorCtrl的派生类处理,也可以被包含我们自定义控件的对话框或者窗口处理.

```
void wxFontSelectorCtrl::OnMouseEvent(wxMouseEvent& event)
{
    if (event.LeftDown())
    {
        // 获取父窗口
        wxWindow* parent = GetParent();
        while (parent != NULL &&
               !parent->IsKindOf(CLASSINFO(wxDialog)) &&
               !parent->IsKindOf(CLASSINFO(wxFrame)))
            parent = parent->GetParent();
        wxFontDialog dialog(parent, m_fontData);
        dialog.SetTitle(_("Please choose a font"));
        if (dialog.ShowModal() == wxID_OK)
        {
            m_fontData = dialog.GetFontData();
            m_fontData.SetInitialFont(
                          dialog.GetFontData().GetChosenFont());
            Refresh();
            wxFontSelectorCtrlEvent event(
                  wxEVT_COMMAND_FONT_SELECTION_CHANGED, GetId());
            event.SetEventObject(this);
            GetEventHandler()->ProcessEvent(event);
        }
    }
}
```

这个控件没有拦截键盘事件,不过你还是可以通过拦截它们来实现和左键单击相同的动作.你也可以在你的控件上绘制一个虚线框来表明是否其当前拥有焦点,这可以通过wxWindow::FindFocus函数判断,如果你决定这样作,你就需要拦截EVT_SET_FOCUS和EVT_KILL_FOCUS事件来在合式的时候进行窗口刷新.

定义默认事件处理函数

如果你看过了wxTextCtrl的实现代码,比如src/msw/textctrl.cpp中的代码,你就会发现一些标准的标识符比如 wxID_COPY,wxID_PASTE, wxID_UNDO和wxID_REDO以及用户界面更新事件都有默认的处理函数.这意味着如果你的应用程序设置了将事件首先交给活动控件处理(参考第20章:"优化你的程序"),这些标准菜单项事件或者工具按钮事件将会拥有自动处理这些事件的能力.当然我们自定义的控件还没有复杂到这种程度,不过你还是可以通过这种机制实现撤消/重做操作以及剪贴板相关操作.我们来看看wxTextCtrl的例子:

```
BEGIN_EVENT_TABLE(wxTextCtrl, wxControl)
    ...
    EVT_MENU(wxID_COPY, wxTextCtrl::OnCopy)
    EVT_MENU(wxID_PASTE, wxTextCtrl::OnPaste)
    EVT_MENU(wxID_SELECTALL, wxTextCtrl::OnSelectAll)

    EVT_UPDATE_UI(wxID_COPY, wxTextCtrl::OnUpdateCopy)
    EVT_UPDATE_UI(wxID_PASTE, wxTextCtrl::OnUpdatePaste)
    EVT_UPDATE_UI(wxID_SELECTALL, wxTextCtrl::OnUpdateSelectAll)
    ...
END_EVENT_TABLE()
void wxTextCtrl::OnCopy(wxCommandEvent& event)
{
    Copy();
}
void wxTextCtrl::OnPaste(wxCommandEvent& event)
{
    Paste();
}
void wxTextCtrl::OnSelectAll(wxCommandEvent& event)
{
    SetSelection(-1, -1);
}
void wxTextCtrl::OnUpdateCopy(wxUpdateUIEvent& event)
{
    event.Enable( CanCopy() );
}
void wxTextCtrl::OnUpdatePaste(wxUpdateUIEvent& event)
{
    event.Enable( CanPaste() );
}
void wxTextCtrl::OnUpdateSelectAll(wxUpdateUIEvent& event)
{
    event.Enable( GetLastPosition() &gt; 0 );
}
```

实现验证器

正如我们在第9章,"创建自定义的对话框"中看到的那样,验证器是数据在变量和控件之间传输的一种很方便地手段.当你创建自定义控件的时候,你可以考虑创建一个特殊的验证器以便应用程序可以使用它来和你的控件进行数据传输.

wxFontSelectorValidator是我们为wxFontSelectorCtrl控件事件的验证器,你可以将其和一个字体指针和颜色指针或者一个wxFontData对象绑定.这些变量通常是在对话框的成员变量中声明的,以便对话框持续跟踪控件改变并且在对话框被关闭以后可以通过其成员返回相应的值.注意验证器的使用方式,不需要使用new函数创建验证器,SetValidator函数将创建一份验证器的拷贝并且在需要的时候自动释放它.如下所示:

```
wxFontSelectorCtrl* fontCtrl =
   new wxFontSelectorCtrl( this, ID_FONTCTRL,
               wxDefaultPosition, wxSize(100, 40), wxSIMPLE_BORDER );
// 或者使用字体指针和颜色指针作为参数
fontCtrl->SetValidator( wxFontSelectorValidator(& m_font,
                                                & m_fontColor) );
// ...或者使用wxFontData指针作为参数
fontCtrl->SetValidator( wxFontSelectorValidator(& m_fontData) );
```

m_font和m_fontColor变量(或者m_fontData变量)将反应用户对字体预览控件所做的任何改变.数据传输在控件所在的对话框的transferDataFromWindow函数被调用的时候发生(这个函数将被默认的wxID_OK处理函数调用).

实现验证器必须的函数包括默认构造函数,带参数的构造函数,一个Clone函数用于复制自己.Validate函数用于校验数据并在数据非法的时候显示相关信息.transferToWindow和transferFromWindow则实现具体的数据传输.

下面列出了wxFontSelectorValidator的声明:

```
/*!
 * wxFontSelectorCtrl验证器
 */
class wxFontSelectorValidator: public wxValidator
{
DECLARE_DYNAMIC_CLASS(wxFontSelectorValidator)
public:
    // 构造函数
    wxFontSelectorValidator(wxFontData *val = NULL);
    wxFontSelectorValidator(wxFont *fontVal,
                            wxColour* colourVal = NULL);
    wxFontSelectorValidator(const wxFontSelectorValidator& val);
    // 析构函数
    ~wxFontSelectorValidator();
    // 复制自己
    virtual wxObject *Clone() const
    { return new wxFontSelectorValidator(*this); }
    // 拷贝数据到变量
    bool Copy(const wxFontSelectorValidator& val);
    // 在需要校验的时候被调用
    // 此函数应该弹出错误信息
    virtual bool Validate(wxWindow *parent);
    // 传输数据到窗口
    virtual bool TransferToWindow();
    // 传输数据到变量
    virtual bool TransferFromWindow();
    wxFontData* GetFontData() { return m_fontDataValue; }
DECLARE_EVENT_TABLE()
protected:
    wxFontData*     m_fontDataValue;
    wxFont*         m_fontValue;
    wxColour*       m_colourValue;
    // 检测是否验证器已经被正确设置
    bool CheckValidator() const;
};
```

建议阅读fontctrl.cpp文件中相关的函数实现以便对齐有进一步的了解.

实现资源处理器

如果你希望你自定义的控件可以在XRC文件中使用,你可以提供一个对应的资源处理器.我们在这里不对此作过多的介绍,请参考第9章,介绍XRC体系时候的相关介绍.同时也可以参考wxWidgets发行目录include/wx/xrc和src/xrc中已经的实现.

你的资源处理器在应用程序中登记以后,XRC文件可以包含你的自定义控件了,它们也可以被你的应用程序自动加载.不过如果制作这个XRC 文件也称为一个麻烦事.因为通常对话框设计工具都不支持动态加载自定义控件.不过通过DialogBlocks的简单的自定义控件定义机制,你可以

指定你的自定义控件的名称和属性,以便生成正确的XRC文件,虽然,DialogBlocks只能作到在其设计界面上显示一个近似的图形以代替你的自定义控件.

检测控件显示效果

当创建自定义控件的时候,你需要给wxWidgets一些关于控件外观的小提示.记住,wxWidgets总会尽可能的使用系统默认的颜色和字体,不过也允许应用程序或者自定义控件在平台允许的时候自己决定这些属性.wxWidgets也会让应用程序或者控件自己决定是否这些属性应该被其子对象继承.控制这些属性的体系确实有一些复杂,不过,除非要大量定制控件的颜色(这是不推荐的)或者实现完全属于自己的控件,程序员很少需要关心这些细节.

除非显式指明,应用程序通常会允许子窗口(其中可能包含你自定义的控件)继承它们父窗口的前景颜色和字体.然后这种行为可以通过使用 SetOwnFont函数设置字体或者使用SetOwnForegroundColour函数设置前景颜色来改变.你的控件也可以通过调用ShouldInheritColours函数来决定是否要继承父窗口的颜色(wxControl默认需要,而wxWindow则默认不需要).背景颜色通常不需要显式继承,你的控件应该通过不绘制不需要的区域的方法来保持和它的父窗口一致的背景.

为了实现属性继承,你的控件应该在构造函数中调用InheritAttributes函数,依平台的不同,这个函数通常可以在构造函数调用wxControl::Create函数的时候被调用.

某些类型实现了静态函数GetClassDefaultAttributes,这个函数返回一个wxVisualAttributes对象,包含前景色,背景色以及字体设定.这个函数包含一个只有在Mac OS X平台上才有意义的参数wxWindowVariant.这个函数指定的相关属性被用在类似GetBackgroundColour这样的函数中作为应用未指定值时候的返回值.如果你不希望默认的值被返回,你可以在你的控件中重新实现这个函数.同时你也需要重载GetDefaultAttributes虚函数,在其中调用GetClassDefaultAttributes函数以便允许针对某个特定的对象返回默认属性.如果你的控件包含一个标准控件的类似属性,你可以直接使用它,如下所示:

```
// 静态函数,用于全局访问
static wxVisualAttributes GetClassDefaultAttributes(
                wxWindowVariant variant = wxWINDOW_VARIANT_NORMAL)
{
    return wxListBox::GetClassDefaultAttributes(variant);
}
// 虚函数,用户对象返回访问
virtual wxVisualAttributes GetDefaultAttributes() const
{
    return GetClassDefaultAttributes(GetWindowVariant());
}

wxVisualAttributes的结构定义如下:

struct wxVisualAttributes
{
    // 内部标签或者文本使用的字体
    wxFont font;
    // 前景色
    wxColour colFg;
    // 背景色; 背景不为纯色背景时可能为wxNullColour
    wxColour colBg;
};
```

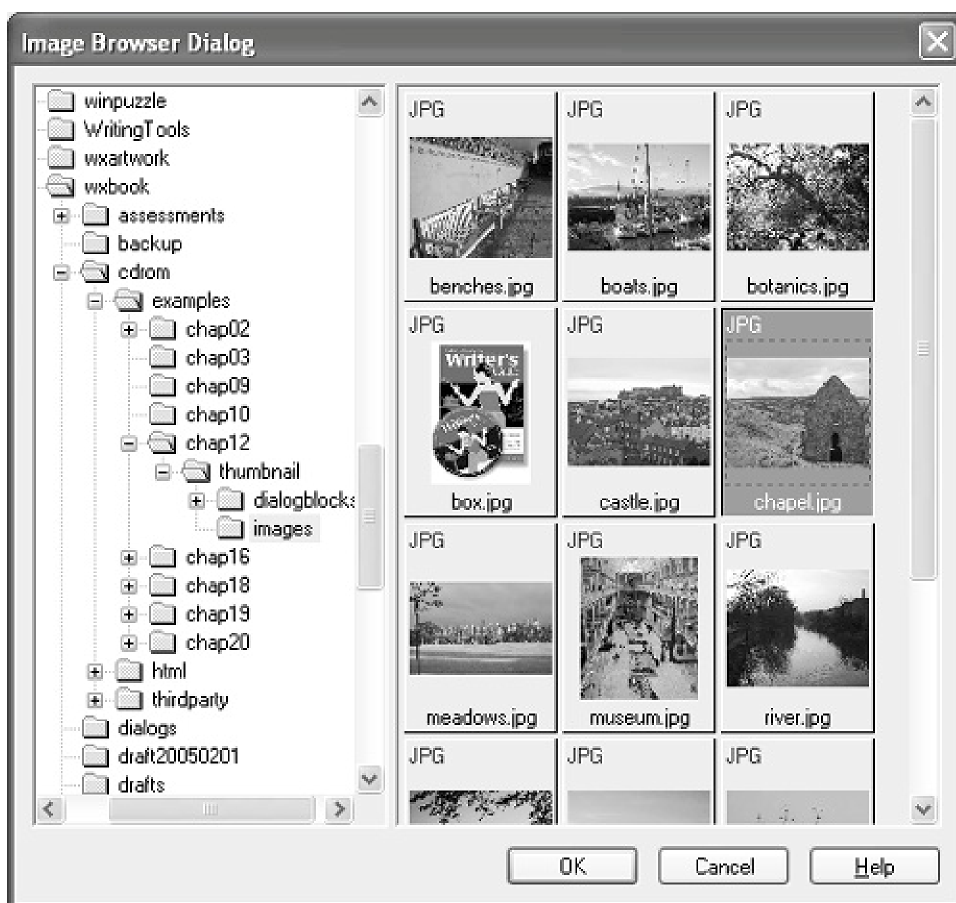
如果你的自定义控件使用了透明背景,比如说,它是一个类似静态文本标签的控件,你应该提供一个函数HasTransparentBackground以便wxWidgets了解这个情况(目前仅支持windows).

最后需要说明的是,如果某些操作需要在某些属性已经确定或者需要在最后的步骤才可以运行.你可以使用空闲时间来作这种处理,在第17章,"多线程编程"的"多线程替代方案"中对此有进一步的描述.

一个更复杂一点的例子:wxThumbnailCtrl

前面我们演示的例子wxFontSelectorCtrl是一个非常简单的例子,很方便我们逐一介绍自定义控件的一些基本原则,比如事件,验证器等.然后,对于显示以及处理输入方面则显得有些不足.在随书光盘的examples/chap12/thumbnail目录中演示了一个更复杂的例子wxThumbnailCtrl,这个控件用来显示缩略图.你可以在任何应用程序中使用它来显示图片的缩略图.(事实上它也可以显示一些别的缩略图,你可以实现自己的wxThumbnailItem的派生类以便使其可以支持显示某种文件类型的缩略图,比如显示那些包含图片的文档中的缩略图).

下图演示的wxThumbnailBrowserDialog对话框使用了wxGenericDirCtrl控件,这个控件使用了wxThumbnailCtrl控件.出于演示的目的,正在显示的这个目录放置了一些图片.



这个控件演示了下面的一些主题,当然列出的并不是全部:

- 鼠标输入: 缩略图子项可以通过单击鼠标左键进行选择或者通过按着Ctrl键单击鼠标左键进行多选.
- 键盘输入: 缩略图网格可以通过键盘导航,也可以通过方向键翻页,子项可以通过按住Shift

键进行选择.
- 焦点处理: 设置和丢失焦点将导致当前的活动缩略图的显示被更新.
- 优化绘图: 通过wxBufferedPaintDC以及仅更新需要更新的区域的方法实现无闪烁更新当前显示.
- 滚动窗口: 这个控件继承自wxScrolledWindow窗口,可以根据子项的数目自动调整滚动条.
- 自定义事件: 在选择,去选择以及右键单击的时候产生wxThumbnailEvent类型的事件.

为了灵活处理,wxThumbnailCtrl并不会自动加载一个目录中所有的图片,而是需要通过下面的代码显式的增加wxThumbnailItem对象.如下所示:

```
// 创建一个多选的缩略图控件
wxThumbnailCtrl* imageBrowser =
   new wxThumbnailCtrl(parent, wxID_ANY,
         wxDefaultPosition, wxSize(300, 400),
         wxSUNKEN_BORDER|wxHSCROLL|wxVSCROLL|wxTH_TEXT_LABEL|
         wxTH_IMAGE_LABEL|wxTH_EXTENSION_LABEL|wxTH_MULTIPLE_SELECT);
// 设置一个漂亮的大的缩略图大小
imageBrowser->SetThumbnailImageSize(wxSize(200, 200));
// 在填充的时候不要显示
imageBrowser->Freeze();
// 设置一些高对比的颜色
imageBrowser->SetUnselectedThumbnailBackgroundColour(*wxRED);
imageBrowser->SetSelectedThumbnailBackgroundColour(*wxGREEN);
// 从'path'路径查找图片并且增加
wxDir dir;
if (dir.Open(path))
{
    wxString filename;
    bool cont = dir.GetFirst(&filename, wxT("*.*"), wxDIR_FILES);
    while ( cont )
    {
        wxString file = path + wxFILE_SEP_PATH + filename;
        if (wxFileExists(file) && DetermineImageType(file) != -1)
        {
            imageBrowser->Append(new wxImageThumbnailItem(file));
        }
        cont = dir.GetNext(&filename);
    }
}
// 按照名称排序
imageBrowser->Sort(wxTHUMBNAIL_SORT_NAME_DOWN);
// 标记和选择第一个缩略图
imageBrowser->Tag(0);
imageBrowser->Select(0);
// 删除第二个缩略图
imageBrowser->Delete(1);
// 显示图片
imageBrowser->Thaw();
```

如果你完整的阅读thumbnailctrl.h和thumbnail.cpp中的代码,你一定会得到关于自定义控件的足够的知识.另外,你也可以在你的应用程序中直接使用wxThumbnailCtrl控件,不要客气.

第十二章小结

本章介绍了一些复杂一点的可视控件,这些控件可能你在第一次使用wxWidgets的时候并不会用到,但是随着你的应用程序复杂程序的增加,你几乎肯定会在你的代码中使用它们中的一个或者多个.最后一小节介绍了怎样实现自定义的控件,结合本章介绍的这些控件的源代码,你可以获得足够多这方面的提示.

同时,你可以考虑参考附录D,"wxWidgets提供的其它特性",里面介绍了更多wxWidgets自带的复杂控件.以及附录E,"wxWidgets的第三方工具",其中介绍了一些第三方的控件.

接下来,我们将介绍wxWidgets中使用的一些数据结构及其类型.

第十三章数据结构类

存储和处理数据是所有应用程序必须的一环.从最简单的用于保存大小和位置信息的类,到复杂的数据类型比如链表和哈希表等,wxWidgets提供了一整套可选的数据结构类.本章将展示很多数据结构类,并着重展示每种数据结构类最常用的函数及方法.对于那些比较少用到的数据结构类及其方法,请参考 wxWidgets的相关文档.

另外,这本书将不会涉及这些数据结构的内部实现,尽管如此,就算不知道其内部是怎样实现的,每个人也都需要知道这些结构应该怎样被使用.

13.1 为什么没有使用STL?

首先,我们来回答一个关于wxWidgets的数据结构类问的最多的一个问题:"为什么它们不采用基于STL(标准模板库)的实现?".最主要的原因是历史原因:wxWidgets从1992年就存在了,这比可以稳定而可靠的支持跨平台交叉编译的STL库要早很久.不过随着wxWidgets的发展,它的许多数据结构类已经拥有了一个和标准STL非常相似的API,希望有一天,wxWidgets中的某些数据结构类可以实现完全的STL兼容.

尽管这样,你还是可以在你的wxWidgets应用程序中使用STL,这需要你将setup.h中的wxUSE_STL置为1(或者在配置wxWidgets的时候使用enable-stl选项),以便使得wxString和别的容器类使用等价的STL实现.不过需要事先声明,在 wxWidgets中允许STL将加大wxWidgets库的大小,并且将延长wxWidgets的编译时间,尤其在使用GCC的时候更加明显.

13.2 字符串类型

使用字符串类来代替标准的字符串指针的好处是被普遍接受的.而wxWidgets就提供了它自己的字符串类:wxString,无论在wxWidgets 内部还是在其提供的API接口上,这个类都被很广泛的使用.wxString类拥有你对一个字符串类期待的所有的操作,包括:动态内存管理,从其它字符串类型构建,赋值操作,单个字符访问,字符串连接和比较,子字符串获取,大小写转换,空格字符的修剪和补齐,查找和替换,类似C语言printf的操作以及类似流一样的插入函数等等.

除了上述的这些字符串处理常用功能,wxString还支持一些额外的特性.wxString完美支持Unicode,包括ANSI字符和Unicode的互相转换,这个特性是和wxWidgets的编译配置无关的.使用wxString还使得你的代码拥有直接将字符串传递给库函数以及直接从库函数返回字符串的能力.另外,wxString已经实现了90%的STL中的std::string类的函数,这意味着对STL熟悉的用户基本上不需要重新学习wxString的使用方法.

使用wxString

在你的应用程序中使用wxString类型是非常简单而直接的.将你程序中使用std::string或者是别的你习惯的字符串类的地方,全部用wxString代替基本就可以了.要注意的是,所有参数中使用字符串的地方,最好使用const wxString&这样的声明(这使得函数内部对于字符串的赋值操作由于使用了引用记数器技术而变得更快速),而所有返回值中使用的字符串则最好直接使用wxString类型,这使得在函数内部即使返回一个局部变量也是安全的.

C和C++的程序员通常都已经很熟悉字符串的各种操作了,因此wxString的详细的API参考就不在这里罗嗦了,请参考wxWidgets的相关文档.

你可能会注意到wxString的好多函数具有同样的功能,比如Length,Len和length函数都返回这个字符串的长度.在这种情况下,你最好使用标准STL兼容的函数形式.这会让你的代码对别的程序员来说更亲切,并且将使你的代码更容易转换为别的不使用wxWidgets库的代码,你甚至可以直接使用typedef将wxString重定义为std::string.另外wxWidgets某一天可能会开始使用标准的std:: string,因此这种作法也会让你的代码更容易保持前向兼容.(当然,wxString的函数也会保留以保证后向兼容.)

wxString,字符以及字符串常量

wxWidgets定义了一个wxChar类型,在不同的编译选项(ANSI或Unicode)下,这个类型用来映射char类型或者 wchar_t类型.象前面提到的那样,你不必使用单独的char类型或者wchar_t类型,wxString内部存储数据的时候使用的是相应的C类型.在任何时候,如果你需要对单个字符进行操作,你应该使用wxChar类型,这将使得你的代码在ANSI版本和Unicode版本中保持一致,而不必使用大量的预定义宏.

如果wxWidgets被编译成Unicode的版本,和标准字符串常量是不兼容的,因为标准字符串常量无论在哪种版本中都是 char*类型.如果你想在Unicode版本中直接使用字符串常量,你应该使用一个转义宏L.wxWidgets提供了一个宏wxT(或者_T)来封装字符串常量,这个宏在ANSI版本中被定义为什么事情也不做,而在Unicode版本中则用来封装L宏,因此无论在哪种版本中,你都可以使用下面的方法使用字符串常量:

```
wxChar ch = wxT('*');
wxString s = wxT("Hello, world!");
wxChar* pChar = wxT("My string");
wxString s2 = pChar;
```

关于使用Unicode版本的更详细的信息,请参考第16章:"编写国际化应用程序".

wxString到C指针的转换基础

因为有时候你需要直接以C类型访问wxString的内部数据进行底层操作,wxWidgets提供了几种对应的访问方法:

- mb_str函数无论在ANSI版本 还是Unicode版本都返回一个const char类型的指针*const char*, 如果是Unicode版本,则字符串首先经过转换,转换过程可能导致数据丢失.
- wc_str函数无论在ANSI版本还是Unicode版本都返回一个wchar_t*类型,如果是ANSI版本,则字符串首先被转换成Unicode版本然后再返回.
- c_str则返回一个指向内部数据的指针 (ANSI版本为const char类型*, Unicode*版本为*const wchar_t*类型).不进行任何转换.

你可以使用c_str函数的特性实现wxString和std::string之间的转换,如下所示:

```
std::string str1 = wxT("hello");
wxString str2 = str1.c_str();
std::string str3 = str2.c_str();
```

使用wxString经常遇到的一个陷井是过度使用对const char*类型的隐式的类型强制转换.我们建议你在任何需要使用这种转换的时候,显式使用c_str来指明这种转换,下面的代码演示了两个常见的错误:

```
// 这段代码将输入的字符串转换为大写函数,然后将其打印在屏幕上
// 并且返回转换以后的值 (这是一个充满bug的代码)
const char *SayHELLO(const wxString& input)
{
    wxString output = input.Upper();
    printf("Hello, %s!\n", output);
    return output;
}
```

上面这四行代码有两个危险的缺陷,第一个是对printf函数的调用.在类似puts这样的函数中,隐式的类型强制转换是没有问题的,因为puts声明其参数为const char*类型,但是对于printf函数,它的参数采用的是可变参数类型,这意味着上述printf代码的执行结果可能是任何一个种结果(包括

正确打印出期待结果),不过最常见的一种结果是程序异常退出,因此,应该使用下面的代码代替上面的printf语句:

```
printf(wxT("Hello, %s!\n"), output.c_str());
```

第二个错误在于函数的返回值.隐式类型强制转换又被使用了以此,因为这个函数的返回值是const char*类型,这样的代码编译是没有问题的,但是它返回的将是一个局部变量的内部指针,而这个局部变量在函数返回以后就很快被释放了,因此返回的指针将变成一个无效指针.解决的方法很简单,应该将返回类型更改为wxString类型,下面列出了修改了以后的代码:

```
// 这段代码将输入的字符串转换为大写函数,然后将其打印在屏幕上
// 并且返回转换以后的值 (这是正确的代码)
wxString SayHELLO(const wxString& input)
{
    wxString output = input.Upper();
    printf(wxT("Hello, %s!\n"), output.c_str());
    return output;
}
```

标准C的字符串处理函数

因为大多数的应用程序都要处理字符串,因此标准C提供了一套相应的函数库.不幸的是,它们中的一部分是有缺陷的(比如strncpy函数有时候不会添加结束符NULL),另外一部分则可能存在缓冲区溢出的危险.而另一方面,一些很有用的函数却没能够进入标准的C函数库.这些都是为什么wxWidgets要提供自己的额外的全局静态函数的原因,wxWidgets的一些静态函数视图避免这些缺陷:wxIsEmpty函数增加了对字符串是否为NULL的校验, 在这种情况下也返回True.wxStrlen函数也可以处理NULL指针,而返回0.wxStricmp函数则是一个平台相关的大小写敏感字符串比较函数,它在某些平台上使用stricmp函数而在另外一些平台上则使用strcasecmp函数.

"wx/string.h"头文件中定义了wxSnprintf函数和wxVsnprintf,你应该使用它们代替标准的 sprintf函数以避免一些sprintf函数先天的危险.带"n"的函数使用了snprintf函数,这个函数在可能的时候对缓冲区进行大小检查.你还可以使用wxString::Printf而不必担心遭受可能受到的针对printf的缓冲区溢出攻击.

和数字的相互转换

应用程序经常需要实现数字和字符串之间的转换,比如将用户的输入转换成数字或者将计算的结果显示在用户街面上.

ToLong(long* val, int base=10)函数可以将字符串转换成一个给定进制的有符号整数.它在成功的时候返回True并将结果保存在val中,如果返回值是False,则表明字符串不是一个有效的对应的进制的数字.指定的进制必须是2到36的整数,0意味着根据字符串的前导符决定: 0x开头的字符串被认为是16进制的, 0-则被认为是8进制的, 其它情况下认为是10进制的.

ToULong(unsigned long* val, int base=10)的工作模式和ToLong函数一致,不过它的转换结果为无符号类型.

ToDouble(double* val)则实现字符串到浮点数的转换.返回值为Bool类型.

Printf(const wxChar* pszFormat, ...) 和C语言的sprintf函数类似,将格式化的文本作为自己的内容.返回值为填充字符串的长度.

静态函数Format(const wxChar* pszFormat, ...)则将格式化的字符串作为返回值.因此你可以使用下面的代码:

```
int n = 10;
wxString s = "Some Stuff";
s += wxString::Format(wxT("%d"),n );
```

操作符"<<"可以用来在wxString中添加一个int,float或者是double类型的值.

wxStringTokenizer

wxStringTokenizer帮助你将一个字符串分割成几个小的字符串,它被用类代替和扩展标准C函数strtok,它的使用方法是:传递一个字符串和一个可选的分割符(默认为空白符),然后循环调用GetNextToken函数直到HasMoreTokens返回False,如下所示:

```
wxStringTokenizer tkz(wxT("first:second:third:fourth"), wxT(":"));
while ( tkz.HasMoreTokens() )
{
    wxString token = tkz.GetNextToken();
    // 处理单个字符串
}
```

默认情况下,wxStringTokenizer对于全空字符串的处理和strtok的处理相同,但是和标准函数不同的是,如果分割符为非空字符,它将把空白部分也作为一个子字符串返回.这对于处理那些格式化的表格数据(每一行的列数相同但是单元格数据可能为空)是比较有好处的,比如使用 tab或者逗号作为分割符的情况.

wxStringTokenizer的行为还受最后一个参数的影响,相关的描述如下:

- wxTOKEN_DEFAULT: 如前所述的默认处理方式; 如果分割符为空白字符则等同于wxTOKEN_STRTOK,否则等同于wxTOKEN_RET_EMPTY.
- wxTOKEN_RET_EMPTY: 在这种模式,空白部分将作为一个子字符串部分被返回,例如"a::b:"如果用":"分割则返回三个子字符串a, ""和b.
- wxTOKEN_RET_EMPTY_ALL: 在这种模式下,最后的空白部分也将作为一个子字符串返回. 这样"a::b:"使用":"分割将返回四个子字符串,其三个和wxTOKEN_RET_EMPTY返回的相同,最后一个则为一个"".
- wxTOKEN_RET_DELIMS: 在这种模式下,分割符也作为子字符串的一部分(除了最后一个子字符串,它是没有分割符的),其它方面类似wxTOKEN_RET_EMPTY.
- wxTOKEN_STRTOK: 这种情况下,子字符串的产生结果和标准strtok函数完全相同.空白字

符串将不作为一个子字符串.

wxStringTokenizer还有下面两个有用的成员函数:

- CountTokens函数返回分割完的子字符串的数目.
- GetPosition返回某个位置的子字符串.

wxRegEx

wxRegEx类用来实现正则表达式.这个类支持的操作包括正则表达式查找和替换.其实现方式有基于系统正则表达式库(比如现代的类 Unix系统以及Mac OSX支持的POSIX标准正则表达式库)或者基于由Henry Spencer提供的wxWidgets内建库.POSIX定义的正则表达式有基础和扩展两套版本.内建的版本支持这两种模式而基于系统库的版本则不支持扩展模式.

即使是对于那些支持正则表达式库的系统,wxWidgets默认的Unicode版本也采用了内建的正则表达式版本,ANSI版本则使用系统提供的版本.记住只有内建版本的正则表达式库才能完全支持Unicode.当编译wxWidgets的时候,覆盖这种默认设置是被允许的.如果在使用系统正则表达式库的Unicode版本中,在使用对应函数的之前,表达式和要匹配的数据都将被转换成8-bit编码的Unicode方式.

使用wxRegEx的方法和其它所有使用正则表达式的方法没有区别.因为正则表达式的内容较为罗嗦而且又鉴于正则表达式的只在特定情况下使用,请参考wxWidgets手册中的相关内容了解具体的API.

13.3 wxArray

wxWidgets使用wxArray提供了一种动态的数组结构.和C语言的数组结构类似,对于其数组项的访问时间为一个常量.对然这样,其内存仍然是动态分配的,换句话说,如果其内存不够再增加子项的时候,它将动态分配内存.在提前分配内存的前提下,其增加数组项的时间也大体上是一个常量. wxArray类型还提供了边界检查的功能,在调试版本中,越界访问将会导致断言错误,而在正式版本中,越界访问将不会出现任何提示,而这种访问的结果可能会是一个随机值.

数组类型

wxWidgets提供了三种不同类型的数组.它们都是wxBaseArray的派生类,在它们的数值类型未定义之前是不能直接使用的. 换句话说,你必须使用WX_DEFINE_ARRAY, WX_DEFINE_SORTED_ARRAY和WX_DEFINE_OBJARRAY宏来定义一个它们的派生类才可以使用它们.这种基类的名称分别为 wxArray,wxSortedArray和wxObjArray,但是你应该有这个概念:这三个类其实是不存在的,并没有这样的类,这个名称只是为了用来说明问题用的.

wxArray这种类型,它的派生类可以用来存储整数类型以及指针类型,这种类型永远不会将它的成员按照对象来对待,也就是说:数组中的元素(无论是整数还是指针)从数组移除的时候并不会被释放.还应该提到的是:wxArray类型的成员函数都是内联函数,因此程序的大小和运行效率完全不受其派生类个数的影响.这种类型最大的限制在于,它只能存储整形数据,比如bool,char,int,long和它们的无符号变体或者任何类型的指针.而浮点行或者double型的数据是不可以用wxArray来存储的.

wxSortedArray和wxArray比较,区别在于,当你需要很频繁的数组进行查找操作时,你应该使用前者.它需要你定义一个比较函数,通过这个比较函数,它将把其内部的元素始终按顺序排列,如果你拥有大量数据并且需要频繁查找,那么使用它性能比使用wxArray要好的多.其它方面两者拥有同样的限制,wxSortedArray也只能存储整形数据.

wxObjArray类则将其内部存储的元素按照对象来对待.它知道在元素从数组中移除的时候释放相应的内存(通过调用其析构函数), 并且使用用于拷贝的构造函数实现拷贝.要定义一个这种类型的派生对象需要两个步骤.首先,使用WX_DECLARE_OBJARRAY宏来声明一个wxObjArray,然后包含其实现文件<wx/arrimpl.cpp>并且在完整声明了其数据元素对象的地方调用 WX_DEFINE_OBJARRAY宏.读起来有些绕口,不过我们很快会举一个简单的例子.

wxArrayString

wxArrayString是存储wxString类型的一个很经济有效的类,它拥有和wxArray类完全一致的功能.它占用的空间也要比直接使用C数组类型wxString[]占用的空间要小的多(这是因为它使用了一些直接对wxString类内部进行操作的方法).所有在 wxArray中可以使用的函数都可以在wxArrayString中使用.

这个类使用上也和其它类差别不大,唯一的区别在于不需要象别的类那样使用WX_DEFINE_ARRAY宏,而可以直接在代码中使用. 当一个wxString实例被插入这个数组时,wxArrayString将创建一份这个字符串的拷贝,应该在成功插入以后,你可以放心的释放原来的字符串.一般情况下,你也不需要关心wxArrayString的内存分配问题,它可以自己释放它所占用的所有的内存.

另外注意Item, Last和操作符[]返回的只是引用而不是拷贝,因此,对它们返回值的操作将导致数组内部数据的改变,如下所示:

```
array.Last().MakeUpper();
```

对应的也有一个wxSortedArrayString对象,这个对象中的字符串总是按照子母顺序排序的.在得到对应字符串的Index 时,wxSortedArrayString使用二进制搜索方式,性能很高.因此如果你的程序中插入字符串的操作很少,而对其进行搜索的操作很多,你可以考虑使用这个类.需要注意的是不要调用这个类的Insert和Sort函数,这两个函数可能会搅乱wxSortedArrayString的内部排序.

数组的构造,析构和内存管理

数组对象也是通用的C++对象,也有对应的构造和赋值操作.拷贝一个wxArray对象意味着其内部元素的拷贝而拷贝一个 wxObjArray对象则是直接拷贝这个数组的子项.不过,出于内存效率的考虑,这两个类都没有虚的析构函数.对于wxArray来说这并不是很重要, 因为其析构函数不需要作什么太多的事情,而对于wxObjArray来说,绝对不要通过删除一个指向wxBaseArray类型的指针来释放相应的数组 (译者注:这将导致对应的析构函数不被调用),并且也永远不要从你自己的数组类再派生新的类型(译者注:同样的原因,新类型的释放将导致使用宏定义的类型的析构函数不被调用).

数组的内存自动管理机制也是很简单的:它在开始的时候会预分配一块小的内存(由宏WX_ARRAY_DEFAULT_INITIAL_SIZE指定).当发现不够用的时候,就增加当前内存的50%(但是不超过 ARRAY_MAXSIZE_INCREMENT).Shrink函数可以用来在没有新数据插入的时候释放多余的内存.而Alloc函数可以在你知道总共需要多少内存的时候被调用以便数组对象不那么频繁的进行分配内存的操作.

数组示例:

下面的例子演示了使用数组最复杂的情况,定义和使用针对自定义类型的wxObjArray数组.使用wxArray的基本原则和例子中演示的是相似的,只不过wxArray类永远不需要和自己内部存储的数据发生什么关系.

```
// 我们自定义的数据类型
class Customer
{
public:
    int CustID;
    wxString CustName;
};
// 这一部分的代码可以位于头文件或者源文件(.cpp)文件中
// 用于声明我们的自定义数组:
WX_DECLARE_OBJARRAY(Customer, CustomerArray);
// 增加下面语句的唯一的要求是,自定义的Customer类型已经完全声明
// (对于WX_DECLARE_OBJARRAY来说,前面的声明已经足够了)
// 而且通常它应该被放在源文件中而不是头文件中.
#include &lt;wx/arrimpl.cpp&gt;
WX_DEFINE_OBJARRAY(CustomerArray);
// 用于排序的时候对两个对象进行比较
int arraycompare(Customer** arg1, Customer** arg2)
{
    return ((*arg1)->CustID &lt; (*arg2)->CustID);
}
// 数组测试例子
void ArrayTest()
{
    // 定义一个我们数组的实例
    CustomerArray MyArray;
    bool IsEmpty = MyArray.IsEmpty(); // will be true
    // 创建一些自定义对象实例
    Customer CustA;
    CustA.CustID = 10;
    CustA.CustName = wxT("Bob");
    Customer CustB;
    CustB.CustID = 20;
    CustB.CustName = wxT("Sally");
    Customer* CustC = new Customer();
    CustC->CustID = 5;
    CustC->CustName = wxT("Dmitri");
    // 将其中两个增加到数组中
    MyArray.Add(CustA);
    MyArray.Add(CustB);
    // 将最后一个插入到数组的任意位置.
    // 数组将会产生一个自定义对象的拷贝
    MyArray.Insert(*CustC, (size_t)0);
    int Count = MyArray.GetCount(); // will be 3
    // 如果找不到将返回wxNOT_FOUND
    int Index = MyArray.Index(CustB); // will be 2
    // 依次处理数组中的对象
    for (unsigned int i = 0; i &lt; MyArray.GetCount(); i++)
    {
        Customer Cust = MyArray[i]; // 或者使用MyArray.Item(i);
        // 进行一些处理
    }
    // 按照自己提供的比较函数进行排序
    MyArray.Sort(arraycompare);
    // 移除但不释放A对象
    Customer* pCustA = MyArray.Detach(1);
    // 如果使用Detach函数,我们需要自己释放对象
    delete pCustA;
    // 如果使用Remove函数,就不需要了
    MyArray.RemoveAt(1);
    // Clear函数也不需要
    MyArray.Clear();
    // 数组从来就不会理会我们自己产生的C对象,要自己释放它
    delete CustC;
}
```

13.4 wxList和wxNode

wxList类是一个双向链表,可以用来存储任何类型的数据.wxWidgets需要你显式的定义一个针对某种数据类型的新的类来使用它,以便对存储于其中的数据提供足够的类型检查.wxList类还允许你指定一个索引类型以便进行基本的查找操作(如果你想使用基于结构的快速随机访问,请参考下一节的 wxHashMap类).

wxList使用了一个虚类wxNode.当你定义一个新的wxList派生类的时候,你同时定义了一个派生自wxNodeBase的类,以便对节点提供类型安全检查.节点类最重要的函数包括:GetNext,GetPrevious和GetData.它们的功能显而易见,分别为:获取下一个子项,获取前一个子项以及获取子项的数据.

唯一值得说明的是wxList的删除操作,默认情况下,从链表中移除一个节点并不会导致节点内部数据的释放.你需要调用 DeleteContents函数来改变这种默认的行为,设置数据随着节点一起释放.如果你想清除整个链表并且释放其中的数据,你应该先调用 DeleteContents,参数为True,然后再调用Clear函数.

我们用不着在这里把手册的内容重新粘贴一遍.我们将举一个简单的例子来演示怎样创建你自己的链表类型.注意WX_DECLARE_LIST宏通常应该位于头文件中,而WX_DEFINE_LIST宏通常应该位于源文件中.

```
// 我们将存储于链表的数据类型
class Customer
{
public:
    int CustID;
    wxString CustName;
};
// 通常应该位于头文件中
WX_DECLARE_LIST(Customer, CustomerList);
// 下面的定义应该位于源文件中,并且通常位于所有Customer声明之后
#include &lt;wx/listimpl.cpp&gt;
WX_DEFINE_LIST(CustomerList);
// 用于排序的比较函数
int listcompare(const Customer** arg1, const Customer** arg2)
{
    return ((*arg1)->CustID &lt; (*arg2)->CustID);
}
// 链表操作举例
void ListTest()
{
    // 定义一个我们自定义链表的实例
    CustomerList* MyList = new CustomerList();
    bool IsEmpty = MyList->IsEmpty(); // will be true
    // 创建一些自定义对象实例
    Customer* CustA = new Customer;
    CustA->CustID = 10;
    CustA->CustName = wxT("Bob");
    Customer* CustB = new Customer;
    CustB->CustID = 20;
    CustB->CustName = wxT("Sally");
    Customer* CustC = new Customer;
    CustC->CustID = 5;
    CustC->CustName = wxT("Dmitri");
    // 将其增加到链表中
    MyList->Append(CustA);
    MyList->Append(CustB);
    // 实现随机插入
    MyList->Insert((size_t)0, CustC);
    int Count = MyList->GetCount(); // will be 3
    // 如果找不到,返回wxNOT_FOUND
    int index = MyList->IndexOf(CustB); // will be 2
    // 自定义的节点里包含了我们自定义的类型
    CustomerList::Node* node = MyList->GetFirst();
    // 节点遍历
    while (node)
    {
        Customer* Cust = node->GetData();
        // 进行一些处理
        node = node->GetNext();
    }
    // 返回特定位置的节点
    node = MyList->Item(0);
    // 按照排序函数排序
    MyList->Sort(listcompare);
    // 移除包含某个对象的节点
    MyList->DeleteObject(CustA);
    // 我们需要自己释放这个对象
    delete CustA;
    // 找到包含某个对象的节点
    node = MyList->Find(CustB);
    // 指示内部数据随节点的删除而删除
    MyList->DeleteContents(true);
    // 删除B的节点的时候,B也被释放了
    MyList->DeleteNode(node);
    // 现在调用Clear,所有的节点和其中数据都将被释放
    MyList->Clear();
    delete MyList;
}
```

13.5 wxHashMap

wxHashMap类是一个简单的,类型安全的并且效率很不错的哈希映射类,它的接口是标准的STL容器接口的一个子集.实际上,它是在标准的std:: map和非标准的std::hash_map之后才可以设计的.通过用于创建哈希表的宏,你可以选择下面的几种类型及其组合作为哈希表的键类型或者数据类型:int,wxString或void*(任意类型).

有三个用来定义哈希映射类的宏.要定义一个名字为CLASSNAME,键类型为wxString,值类型为VALUE_T类型的哈希表,你可以使用下面的语法:

```
WX_DECLARE_STRING_HASH_MAP(VALUE_T, CLASSNAME);
```

要定义一个名字为CLASSNAME,键类型为void*,值类型为VALUE_T类型的哈希表,使用下面的定义:

```
WX_DECLARE_VOIDPTR_HASH_MAP(VALUE_T, CLASSNAME);
```

要定义一个名称为CLASSNAME,键类型和值类型任意类型的哈希表,使用下面的定义:

```
WX_DECLARE_HASH_MAP(KEY_T, VALUE_T, HASH_T, KEY_EQ_T, CLASSNAME);
```

HASH_T和KEY_EQ_T是用来作为哈希算法和比较算法的函数. wxWidgets提供了三种预定义的哈希算法: wxIntegerHash用来作为整数的哈希算法(int, long, short和它们的无符号变体都可以), wxStringHash用来作为字符串的哈希算法(wxString, wxChar, *char*都可以), wxPointerHash用来作为任何指针类型的哈希算法.类似的也有三个预定义的比较函数: wxIntegerEqual, wxStringEqual和wxPointerEqual.

下面的代码演示了wxHashMap的使用方法:

```
// 我们要存放在哈希表中的类
class Customer
{
    public:
        int CustID;
        wxString CustName;
};
// 定义和实现我们自定义的哈希表.
WX_DECLARE_HASH_MAP(int, Customer*, wxIntegerHash,
                    wxIntegerEqual, CustomerHash);
void HashTest()
{
    // 定义一个自定义哈希表的实例
    CustomerHash MyHash;
    bool IsEmpty = MyHash.empty(); // will be true
    // 创建几个对象
    Customer* CustA = new Customer;
    CustA->CustID = 10;
    CustA->CustName = wxT("Bob");
    Customer* CustB = new Customer;
    CustB->CustID = 20;
    CustB->CustName = wxT("Sally");
    Customer* CustC = new Customer;
    CustC->CustID = 5;
    CustC->CustName = wxT("Dmitri");
    // 将对象增加到哈希表
    MyHash[CustA->CustID] = CustA;
    MyHash[CustB->CustID] = CustB;
    MyHash[CustC->CustID] = CustC;
    int Size = MyHash.size(); // will be 3
    // count函数返回0或1, 含义为:20这个关键值在哈希表中吗?
    int Present = MyHash.count(20); //将返回1
    // 我们哈希表的自定义节点类型
    CustomerHash::iterator i = MyHash.begin();
    // 遍历哈希表
    while (i != MyHash.end())    {
        // first函数返回键值,second返回数据
        int CustID = i->first;
        Customer* Cust = i->second;
        // 作一些处理
        // 然后处理下一个数据
        i++;
    }
    // 将键值为10的数据移出哈希表
    MyHash.erase(10);
    // 移出不会导致数据自动释放
    delete CustA;
    // 返回指定键值的一个节点
    CustomerHash::iterator i2 = MyHash.find(21);
    // 判断是否找到节点
    bool NotFound = (i2 == MyHash.end()); // 将返回True
    // 这次将返回有效的节点
    i2 = MyHash.find(20);
    // 直接移除节点
    MyHash.erase(i2);
    delete CustB;
    // 副作用: 下面语句导致哈希表中插入一个键值为30,值为NULL的节点.
    Customer* Cust = MyHash[30]; // Cust将等于NULL
    // 清除哈希表中的节点
    MyHash.clear();
    delete CustC;
}
```

13.6 存储和使用日期和时间

wxWidgets提供了一个功能强大的类wxDateTime来进行时间和日期相关的操作,包括:格式化输出,时区,时间和日期计算等等.还提供了一些静态函数来提供当前的日期和时间以及查询某个给定的年份是不是闰年等.注意即使你只想操作日期或者是时间,你仍然可以使用wxDateTime类型. wxTimeSpan和wxDateSpan类型提供了修改一个wxDateTime值的合适的方法.

wxDateTime

wxDateTime类拥有很多的成员函数,每个函数的含义都很清晰.完整的API可以参考wxWidgets的相关手册,下面只对其中使用最频繁的函数进行一些介绍.

注意尽管时间在其内部总是以格林威治时间(GMT)存储的,但是你通常关心的是本地时区的时间而不是格林威治时间,因此, wxDateTime的构造函数以及更改函数中各个组成时间的要素(比如小时,分钟和秒钟)等都指的是本地时区的时间.所有用于获取时间和日期的函数返回的要素(月,日,小时,分,秒等)也都是本地时间.因此,如果这是你需要的,你不需要作任何额外的操作,如果你希望操作不同时区的时间,请参考相关的文档.

wxDateTime类的构造和更改

wxDateTime可以通过Unix时间戳,仅包含时间的信息,仅包含日期的信息,完整的时间日期信息等途径创建.对于每一个构造函数,都有一个对应的Set函数用来更改已经设置了值的wxDateTime对象.也可以通过类似SetMonth或者SetHour等函数更改时间或者日期中的某个要素.

wxDateTime(time_t)函数根据一个指定的Unix时间戳来构造对象.

wxDateTime(const struct tm&)函数根据一个指定的C语言标准tm结构构造对象.

wxDateTime(wxDateTime_t hour, wxDateTime_t minute = 0, wxDateTime_t second = 0, wxDateTime_t millisec = 0)根据指定的时间要素构造对象.

wxDateTime(wxDateTime_t day, Month month = Inv_Month, int year = Inv_Year, wxDateTime_t hour = 0, wxDateTime_t minute = 0, wxDateTime_t second = 0, wxDateTime_t millisec = 0) 根据指定的时间和日期要素构造对象

wxDateTime访问方法

大多数wxDateTime类的访问函数都是自解释的,比如: GetYear, GetMonth, Getday, GetWeekDay, GetHour, GetMinute, GetSecond, GetMillisecond, GeTDayOfYear, GetWeekOfYear, GetWeekOfMonth和GetYearDay等. wxDateTime还提供了下列一些访问函数:

- GetTicks返回一个Unix时间戳(也就是自从1970年1月1日午夜以来的秒数).

- IsValid返回时间日期类是不是已经被初始化(类自使用默认构造函数创建以后始终未被赋值).

获取当前时间

wxDateTime提供了两个静态函数返回当前时间:

- wxDateTime::Now 返回精度为秒的当前时间.
- wxDateTime::UNow 返回精度为毫秒的当前时间.

时间和字符串的转换

下面介绍的这些函数用来实现时间和字符串的相互转换.将时间转化成字符串的方法是比较简单的:你可以将时间转换成本地格式的字符串 (FormatDate和FormatTime函数),或者以定义在ISO 8601中的国际标准格式来显示(FormatISODate函数和FormatISOTime函数),也可以以自定义的格式来显示(Format函数).

而从文本到时间的转换则显得更复杂些,因为可能的时间格式太多了.最简单的函数是ParseFormat函数,它用来解析那些指定格式的时间文本.ParseRfc822Date函数用来解析那些定义在RFC822中的时间表示方法,这种方法在email或者互联网上使用比较普遍.

最有趣的文本到时间转换函数是ParseTime,ParseDate和ParseDateTime函数,它们将尽量匹配各种格式的时间文本.除了预定义的那些标准格式以外,ParseDateTime函数甚至可以支持那些类似"tomorrow"(明天), "March first"(三月一日)以及"next Sunday"(下个星期天)这样的时间.

日期比较

两个wxDateTime对象可以通过各种函数进行比较,这些函数都返回bool类型的值.这些函数包括:IsEqualTo, IsEarlierThan, IsLaterThan, IsSameDate和IsSameTime等.

而IsStrictlyBetween和IsBetween则用来比较某个日期是不是在两个日期之间.这两个函数的区别在于,如果要比较的时间刚好等于其中的与其比较的边界值的时候,前者返回False而后者返回True.

日期计算

wxWidgets提供了两个非常灵活的类wxTimeSpan和wxDateSpan来辅助进行日期和时间的计算. wxTimeSpan用来计算那些以毫秒为单位的,跨度不大的,快速的和精确的计算,而wxDateSpan则用来进行跨度比较大的比如几周或者几个月的计算.wxDateSpan将尽可能使用更自然的方法来进行计算,因此其含义有时候并不象它看上去的那样精确.比如1月31日加上一个月将返回二月28日 (或29日),也就是说是二月的最后一天,而不是永远不可能存在的二月31日.通常,你比较喜欢这样的结果,不过有时候相应的减法运算可能也会把你搞糊涂,比如二月28日减去一个月的结果的一月28日而不是一月31日.

日期类型可以进行的操作很多,但是这些操作的组合却未必是有效的.比如:对一个日期进行乘法运算是无效的,而对于任何一个表示时间间隔的类(wxTimeSpan或wxDateSpan)进行乘法运算则没有任何问题.

- 加法: wxTimeSpan或wxDateSpan可以和wxDateTime进行加法运算,返回一个新的wxDateTime对象. 两个相同类型的时间间隔类也可以进行加法运算,返回一个新的同样类型的对象.
- 减法: 减法适用和加法同样的规则,额外的一个规则是两个wxDateTime对象相减返回一个wxTimeSpan对象.
- 乘法: wxTimeSpan或wxDateSpan对象可以乘以一个整数,返回一个同样类型的对象.
- Unary相减:wxTimeSpan或wxDateSpan对象可以定义为负数,导致相反的时间方向上的同样的间隔.

下面的例子演示了wxDateSpan和wxTimeSpan的用法,更多的用法请参考wxWidgets的相关手册.

```
void TimeTests()
{
    // 获取当前时间和日期
    wxDateTime DT1 = wxDateTime::Now();
    // 创建一个2星期零1天,或者说15天的间隔
    wxDateSpan Span1(0, 0, 2, 1);
    // 今天减去15天
    wxDateTime DT2 = DT1 - Span1;
    // 用静态方法创建一天的间隔
    wxDateSpan Span2 = wxDateSpan::Day();
    // Span3将代表14天的间隔
    wxDateSpan Span3 = Span1 - Span2;
    // 0 天 (这个间隔将用2周来表示)
    int Days = Span3.GetDays();
    // 14 天 (2 周)
    int TotalDays = Span3.GetTotalDays();
    // 之前两周
    wxDateSpan Span4 = -Span3;
    // 3个月的间隔
    wxDateSpan Span5 = wxDateSpan::Month() * 3;
    // 10小时5分6秒的间隔
    wxTimeSpan Span6(10, 5, 6, 0);
    // DT2增加固定的间隔Span6
    wxDateTime DT3 = DT2 + Span6;
    // Span7是相反方向上的3倍Span6的间隔.
    wxTimeSpan Span7 = (-Span6) * 3;
    // SpanNeg将返回True, 这个间隔是负方向的.
    bool SpanNeg = Span7.IsNegative();
    // 适用静态方法创建一个1小时的间隔
    wxTimeSpan Span8 = wxTimeSpan::Hour();
    // 1小时当然小于30小时(这里使用绝对值)
    bool Longer = Span8.IsLongerThan(Span7);
}
```

13.7 其它常用的数据类型

wxWidgets内部使用了一些其它的数据类型,也在一些公用API中作为参数和返回值,并且wxWidgets也鼓励程序员在它们的代码中使用这些类型.

wxObject

wxObject类是所有wxWidgets类的基类,它提供的功能包括运行期类型信息,引用技术,虚析构函数,可选的调试版本的new和delete函数等.某些wxObject对象的成员函数还使用了用于存储meta-data的wxClassInfo对象.

```
MyWindow* window = wxDynamicCast(FindWindow(ID_MYWINDOW), MyWindow);
```

IsKindOf函数判断对象是否是传入的wxClassInfo指向的类型.

```
bool tmp = obj->IsKindOf(CLASSINFO(wxFrame));
```

Ref函数的参数为const wxObject&类型, 它的作用是将当前对象的数据替换为参数对象的引用.当前对象的引用技术减一,如果需要则释放当前对象数据,参数对象的引用技术则加一.

UnRef则将对象内部数据的引用记数器减一,如果已经减到0则释放当前对象数据.

wxLongLong

wxLongLong类用来存储64位整数.如果本地系统支持64位长整数则使用本地系统提供的实现,否则将使用模拟的64位实现.这个类的使用和其它标准的数字类型没有区别.注意它是一个有符号数字,如果想使用它的无符号版本,可以使用wxULongLong类型,后者的API和前者几乎完全一样,除了个别的函数(比如求绝对值函数)可能返回不同的结果.除了一般的计算函数以外,另外的几个常用的函数包括:

- Abs函数返回wxLongLong的绝对值,如果是作为常量引用调用的这个函数,则返回源对象的一个拷贝,否则将修改源对象的内部数值.
- ToLong 函数将其转换成一个长整型,如果由于存在精度丢失,在调试版本中将引发一个断言错误.
- ToString将内部存储的数据转换成一个wxString类型.

wxPoint和wxRealPoint

wxPoint在wxWidgets中使用比较普遍,常用来代表屏幕或者窗口上的一个确定的位置.正如它的名字的意思一样,它内部的数据用x和y两个整数代表一个座标值.其数据成员是以public方式定义的,可以直接被其它对象访问.wxPoint支持和另外一个wxPoint对象或者 wxSize对象进行加

法和减法的运算.wxRealPoint对象和wxPoint对象类似,不过其内部成员是double类型,并且只支持和别的 wxRealPoint类型进行加减运算.

构造wxPoint实例的方法很直接:

```
wxPoint myPoint(50, 60);
```

wxRect

wxRect通常在绘画或者窗口类中使用(比如wxDC或者wxtreeCtrl),用来定义一个矩形区域.其内部的数据成员除了x和y 以外,还包括宽度和高度.所有的成员都是public类型,可以直接被其它的类访问.除了同类型之间的加减运算,wxRect还支持一些其它运算:

GetRight返回矩形最右边的X座标.

GetBottom返回底端的Y座标.

GetSize返回一个wxSize对象用来表征矩形区域的宽度和高度.

Inflate函数增加矩形区域的大小,如果只有一个参数,则长和宽增加一样的大小,如果是两个参数,则长和宽分别增加对应的大小.

Inside函数判断某个点是否位于矩形区域以内,点的格式可以是单独的XY座标,也可以是一个wxPoint类型.

Intersects判断某个矩形是否和另外一个矩形有重叠区域.

Offset将当前矩形偏移一段举例,偏移的位置既可以通过X和Y单独指定,也可以通过wxPoint来指定.

下面的代码演示了wxRect的三种构造函数:

```
wxRect myRect1(50, 60, 100, 200);
wxRect myRect2(wxPoint(50, 60), wxPoint(150, 260));
wxRect myRect3(wxPoint(50, 60), wxSize(100, 200));
```

wxRegion

wxRegion用来代表设备上下文或者窗口上的一个简单的或者复杂的区域.它使用了引用记数,因此拷贝和赋值操作是非常快速的.它的主要用途是用来定义或者查询某个需要裁剪或者更新的区域.

Contains函数在其参数指定的座标, wxPoint, 矩形或 wxRect被包含在区域内时返回True.

GetBox函数返回一个包含整个区域的wxRect对象.

Intersect函数在指定的矩形,wxRect或wxRegion参数和本区域有重叠的时候返回True.

Offset对区域进行指定x和y方向数量的平移.

Subtract, Union和Xor函数提供了一种灵活的机制来改变当前区域.这三个函数的变体函数(函数名相同,参数不同)超过10个.所有这些函数都可以支持wxRegion参数或者wxPoint参数.请参考wxWidgets的相关手册内容.

下面的代码演示了四种创建区域的方法,所有这些方法创建的结果都是一样的区域:

```
wxRegion myRegion1(50, 60, 100, 200);
wxRegion myRegion2(wxPoint(50, 60), wxPoint(150, 260));
wxRect myRect(50, 60, 100, 200);
wxRegion myRegion3(myRect);
wxRegion myRegion4(myRegion1);
```

你可以使用wxRegionIterator类来遍历某个区域中的矩形区域,比如要在窗口重绘函数中只绘制那些需要绘制的矩形区域,你可以使用下面的代码:

```
// 在窗口需要被重绘的时候调用
void MyWindow::OnPaint(wxPaintEvent& event)
{
    wxPaintDC dc(this);
    wxRegionIterator upd(GetUpdateRegion());
    while (upd)
    {
        wxRect rect(upd.GetRect());
        // 刷新这个矩形区域
        ...一些代码...
        upd ++ ;
    }
}
```

wxSize

wxSize类型在wxWidgets广泛用于指定窗口,控件,画布等等对象的大小.很多需要返回大小信息的函数也将返回这个对象类型.

GetHeight和GetWidth函数返回高度和宽度.

SetHeight和SetWidth函数设置整数的高度和宽度.

Set函数则使用两个整数参数来同时改变高度和宽度.

wxSize的创建也非常简单,如下所示:

```
wxSize mySize(100, 200);
```

wxVariant

wxVariant类用来表示那些可以是任意类型的数据.数据的类型甚至可以动态的改变.这种类型在解决某些特定的问题的时候很有用处,比如要编辑不同类型的数据的编辑器或者用于实现远程过程调用.

wxVariant类型可以存储的数据包括bool, char, double, long, wxString, wxArrayString, wxList, wxDateTime, void* 和可变类型变量列表.不过,你还是可以通过实现wxVariantData的派生类的发生扩展wxVariant可以支持的数据类型.在构造和赋值的时候,只需要将其当成wxVariantData使用就可以了.当然,不方便的地方在于如果你要访问自定义的数据类型,需要先将其转换成wxVariantData对象,而不象内置支持的类型那样,有对应的类似于GetLong这样的函数支持.

另外,要记住不是所有的类型都可以互相转换,比如你不可能把一个bool型的数据转换成wxDateTime类型,也不可能把一个整数转换成wxArrayString,你需要按照一些常识来判断哪些数据类型是可以互相转换的,并且你总是可以通过GetType函数来得到当前数据最合适的类型.下面举一个使用wxVariant类的简单的例子:

```
wxVariant Var;
// 存储wxDateTime类型, 获取wxString类型
Var = wxDateTime::Now();
wxString DateAsString = Var.GetString();
// 存储一个wxString类型, 获取一个double类型
Var = wxT("10.25");
double StringAsDouble = Var.GetDouble();
// 当前的类型应该是"string"
wxString Type = Var.GetType();
// 下面演示一个无理取闹的转换
// 由于不能转换,所以转换的结果为0
char c = Var.GetChar();
```

第十三章小结

wxWidgets提供的多种数据结构类型让你可以很容易的从wxWidgets提供的API中或者你自己的应用程序中获取和使用各种类型的结构化数据. 通过功能强大的数据类型wxRegEx,wxStringTokenizer,wxDateTime,wxVariant等等,你几乎不需要使用任何第三方的库就可以处理各种数据.

接下来,我们来看看wxWidgets提供的从文件或者流中读取或者写入数据的方法.

第十四章文件和流操作

在这一章里,我们来看看wxWidgets提供的底层文件操作对象以及流操作.wxWidgets的流对象不但能使得你的应用程序可以和各种标准的C++ 库打交道,而且还提供完整的压缩,写zip文档以及socket读写等操作.我们还将描述一下wxWidgets提供的虚拟文件系统,它让你的应用程序可以很容易的从非常规文件中获取各种资源.

14.1 文件类和函数

wxWidgets提供了一系列的平台无关的文件处理功能.在概览所有文件函数之前,我们先来看看几个主要的类.

wxFile和wxFFile

wxFile类可以用来处理底层的文件输入输出.它封装了常用的用于操作整数文件标识符的标准C操作(打开关闭,读写,移动游标等),但是和标准C不同的是,它使用wxLog类来报告错误并且在析构函数中自动关闭文件.而wxFFile则提供缓冲的输入输出操作,内部使用的是FILE类型的指针.

你可以通过下面的方法创建一个wxFile对象:使用默认构造函数然后调用Create或者Open函数;或者直接在构造函数中指明文件名和打开模式(wxFile::read, wxFile::write或wxFile::read_write);或者直接使用已经存在的整数文件描述符(相关构造函数或者默认构造函数加 Attach函数).Close函数关闭当前使用的文件,文件也将在析构函数中视需要进行关闭.

从文件中读取数据使用Read函数,参数为一个void和一个缓冲区大小,返回实际读取的数值大小或者 *wxInvalidOffset*如果读取过程发生错误.使用*Write*函数将指定大小的*void*类型的缓冲区写入文件,如果你希望写操作立即写到物理文件,你需要使用Flush函数.

Eof函数用来检测当前的文件指针是否位于文件结尾的位置(而wxFFile的Eof函数只有在其读操作越过了文件结尾的时候才返回True).你可以用Length函数返回文件的长度.

Seek和SeekEnd函数将文件指针移动到相对于文件开始或者文件结尾的一个偏移位置.Tell函数返回wxFileOffset类型的当前文件指针位置,这个类型在支持64位操作系统上是64位整数,否则是32位整数.

Access函数是一个静态函数,用来判断某个文件是否可以以指定的模式打开,而Exists函数则用来判断指定的文件是否存在.

下面的代码演示了怎样使用wxFile打开一个文件并且将其内容读取到一个数组中:

```
#include "wx/file.h"
if (!wxFile::Exists(wxT("data.dat")))
    return false;
wxFile file(wxT("data.dat"));
if ( !file.IsOpened() )
    return false;
//文件大小
wxFileOffset nSize = file.Length();
if ( nSize == wxInvalidOffset )
    return false;
// 将所有内容读取到一个数组中
wxUint* data = new wxUint8[nSize];
if ( fileMsg.Read(data, (size_t) nSize) != nSize )
{
    delete[] data;
    return false;
}
file.Close();
```

下面的代码则演示了怎样将一个文本框的所有内容写入到文件中:

```
bool WriteTextCtrlContents(wxTextCtrl* textCtrl,
                           const wxString& filename)
{
    wxFile file;
    if (!file.Open(filename, wxFile::write))
        return false
    int nLines = textCtrl->GetNumberOfLines();
    bool ok = true;
    for ( int nLine = 0; ok && nLine &lt; nLines; nLine++ )
    {
        ok = file.Write(textCtrl->GetLineText(nLine) +
                        wxTextFile::GetEOL());
    }
    file.Close();
    return ok;
}
```

wxTextFile

wxTextFile提供了一种非常直接的方式来以行为单位读取和写入小型的文本文件.

使用Open函数将这个文本文件读取到内存中并且以行为单位进行分割,使用Write函数写回到文本文件.你可以使用GetLine函数或者直接按照数组的方式操作某个特定的行.或者使用GetFirstLine,GetNextLine和GetPrevLine进行遍历.AddLine 和InsertLine用来增加新行,RemoveLine用来移除特定的行,Clear函数则用来清空所有的行.

下面的例子演示了将文本文件中的每一行都增加一个前导文本的方法:

```
#include "wx/textfile.h"
void FilePrepend(const wxString& filename, const wxString& text)
{
    wxTextFile file;
    if (file.Open(filename))
    {
        size_t i;
        for (i = 0; i &lt; file.GetLineCount(); i++)
        {
            file[i] = text + file[i];
        }
        file.Write(filename);
    }
}
```

wxTempFile

wxTempFile是wxFile的一个派生类,它使用临时文件来写入数据,数据在Commit函数被调用之前不会被写入.如果你需要写一些用户数据,你可以将其写在临时文件里,它的好处是:如果遇到突然的断电或者应用程序不可知错误或者其它大的错误时,临时文件不会对磁盘上的文件系统造成任何伤害.

提示:文档/视图框架在创建一个输出流然后调用SaveObject的时候没有使用临时文件.你可以尝试重载其 DoSaveDocument函数,在其中构建一个wxFileOutputStream并且让其使用一个临时文件wxTempFile对象,在全部数据写完以后,调用Sync函数和Commit函数将其写入临时文件.

wxDir

wxDir是一个轻便的等价于Unix上的open/read/closedir函数的类,它支持枚举目录中的所有文件.wxDir既支持枚举目录中的文件,也支持枚举目录中的子目录.它还提供了一个灵活的递归枚举文件的方法Traverse函数,和另外一种简单的方法: GetAllFiles函数.

首先,你需要调用Open函数打开一个目录(或者通过构造函数直接赋值),然后调用GetFirst函数,参数为一个指向字符串类型的指针用来接收找到的文件名,一个可选的文件通配符(默认为所有文件)和一个可选的选项.然后调用GetNext函数直到其返回False.文件通配符可以是固定的文件名以及包含"*(匹配任意字符)"和"?"(匹配单个字符)的通配符.选项参数可选的值为:wxDIR_FILES(所有文件), wxDIR_DIRS(所有文件夹),wxDIR_HIDDEN(隐藏文件)以及wxDIR_DOTDOT("."和"..")以及它们的组合,默认值为除了最后一项的所有文件.

下面是一个例子:

```
#include "wx/dir.h"
wxDir dir(wxGetCwd());
if ( !dir.IsOpened() )
{
    // 如果遇到这个情况,wxDir已经显示了一个出错信息.
    // 所以直接返回就可以了
    return;
}
puts("Enumerating object files in current directory:");
wxString filename;
wxString filespec = wxT("*.*");
int flags = wxDIR_FILES|wxDIR_DIRS;
bool cont = dir.GetFirst(&filename, filespec, flags);
while ( cont )
{
    wxLogMessage(wxT("%s\n"), filename.c_str());
    cont = dir.GetNext(&filename);
}
```

如同上面注释中说的那样,如果wxDir打开的时候出现错误,将会弹出一个错误消息,如果想禁止这个消息,你可以临时通过设置wxLogNull的方法,如下所示:

```
{
    wxLogNull logNull;
    wxDir dir(badDir);
    if ( !dir.IsOpened() )
    {
        return;
    }
}
```

wxFileName

wxFileName用来处理文件名.它可以分解和组合文件名,还提供了很多额外的操作,其中某些为静态函数.下面演示了一些例子,更多的功能请参考wxWidgets的手册:

```
#include "wx/filename.h"
// 使用字符串创建文件名
wxFileName fname(wxT("MyFile.txt"));
// Normalize,在windows平台上这个函数的动作包括
// 确保文件名为长文件名格式
fname.Normalize(wxPATH_NORM_LONG|wxPATH_NORM_DOTS|wxPATH_NORM_TILDE|
                wxPATH_NORM_ABSOLUTE);
// 返回全路径
wxString filename = fname.GetFullPath();
// 返回相对于当前目录的路径
fname.MakeRelativeTo(wxFileName::GetCwd());
// 文件存在吗?
bool exists = fname.FileExists();
// 另外一个文件存在吗?
bool exists2 = wxFileName::FileExists(wxT("c:\\temp.txt"));
// 返回文件名的名称部分
wxString name = fname.GetName();
// 返回路径部分
wxString path = fname.GetPath();
// 在windows系统上返回相应的短路径文件名,其它平台上
// 返回本身.
wxString shortForm = fname.GetShortPath();
// 创建一个文件夹
bool ok = wxFileName::Mkdir(wxT("c:\\thing"));
```

File Functions

下表列出了一些有用的静态文件操作函数,它们定义在头文件wx/filefn.h中.请同时参考wxFileName类,尤其是其中的静态函数部分,比如wxFileName::FileExists函数.

wxDirExists(dir)	**是否目录存在. 参考wxFileName::DirExists**
wxConcatFiles(f1, f2, f3)	将f1和f2合并为f3, 成功时返回True.
wxCopyFile(f1, f2, overwrite)	拷贝f1到f2,可选择是否覆盖已存在的f2.返回Bool型
wxFileExists(file)	测试是否文件存在. 参考wxFileName::FileExists
wxFileModificationTime(file)	返回文件修改时间(time_t类型). 参考wxFileName::GetModificationTime,它返回wxDateTime类型
wxFileNameFromPath(file)	返回文件全路径的文件名部分. 推荐使用wxFileName::SplitPath函数
wxGetCwd()	返回当前工作目录. 参考wxFileName::GetCwd
wxGetdiskSpace (path, total, free)	返回指定路径所在的磁盘的全部空间和空闲空间,空间为wxLongLong类型.
wxIsAbsolutePath(path)	测试指定的路径是否为绝对路径.
wxMkdir(dir, permission=777)	创建一个目录,其父目录必须存在,可选指定目录访问掩码.返回bool型
wxPathOnly(path)	返回给定全路径的目录部分.
wxRemoveFile(file)	删除文件,成功时返回True.
wxRenameFile(file1, file2)	重命名文件,成功时返回True.如果需要进行文件拷贝,则直接返回False.
wxRmdir(file)	移除空目录,成功时返回True.
wxSetWorkingDirectory(file)	设置当前工作目录,返回bool型.

wxWidgets同样提供了许多函数来封装标准C函数,比如: wxFopen, wxFputc和wxSscanf,这些函数没有在手册中记录,不过你应该可以在include/wx/wxchar.h中找到它们.

另外一个有用的宏是wxFILE_SEP_PATH,它代表了不同平台上的路径分割符,比如在windows平台上它代表"\",而在Unix平台上则代表"/".

14.2 流操作相关类

所谓流模型，指的是一种用于提供相对于文件读写更高层的数据读写的模型.使用流模型,你的代码不需要关心当前操作的是文件,内存还是socket(参考第 18章,"使用wxSocket编程",其中演示了用流方式使用socket的方法).某些wxWidgets标准类同时支持文件读写操作和流读写操作,比如wxImage类.

wxStreamBase类是所有流类的基类,它声明的函数包括类似OnSysRead和OnSysWrite等需要继承类实现的函数.其子类wxInputStream和wxOutputStream则提供了更具体的流类(比如wxFileInputStream和 wxFileOutputStream子类)共同需要的用于读写操作的基本函数.让我们来具体来看一下wxWidgets提供的各种流操作相关类.

文件流

wxFileInputStream和wxFileOutputStream是基于wxFile类实现的,可以通过文件名,wxFile 对象或者整数文件描述符的方式来进行初始化.下面的例子演示了使用wxFileInputStream来进行文件读取,文件指针移位并读取当前位置数据等.

```
#include "wx/wfstream.h"
// 构造函数初始化流缓冲区并且打开文件名参数对应的文件.
// wxFileInputStream将在析构函数中自动关闭对应的文件描述符.
wxFileInputStream inStream(filename);
// 读100个字节
int byteCount = 100;
char data[100];
if (inStream.Read((void*) data, byteCount).
             LastError() != wxSTREAM_NOERROR)
{
    // 发生了异常
    // 异常列表请参考wxStreamBase的相关文档.
}
// 你可以通过下面的方法判断到底成功读取了多少个字节.
size_t reallyRead = inStream.LastRead();
// 将文件游标移动到文件开始处.
// SeekI函数的返回值为移动游标以前的游标相对于文件起始处的位置
off_t oldPosition = inStream.SeekI(0, wxFromBeginning);
// 获得当前的文件游标位置
off_t position = inStream.TellI();
```

使用wxFileOutputStream的方法也很直观.下面的代码演示了使用wxFileInputStream和wxFileOutputStream实现文件拷贝的方法.每次拷贝1024个字节.为了使代码更简介,这里没有显示错误处理的代码.

```
// 下面的代码实现固定单位大小的流拷贝.
// 缓冲区的使用是为了加快拷贝的速度.
void BufferedCopy(wxInputStream& inStream, wxOutputStream& outStream,
                  size_t size)
{
    static unsigned char buf[1024];
    size_t bytesLeft = size;
    while (bytesLeft &gt; 0)
    {
        size_t bytesToRead = wxMin((size_t) sizeof(buf), bytesLeft);
        inStream.Read((void*) buf, bytesToRead);
        outStream.Write((void*) buf, bytesToRead);
        bytesLeft -= bytesToRead;
    }
}
void CopyFile(const wxString& from, const wxString& to)
{
    wxFileInputStream inStream(from);
    wxFileOutputStream outStream(to);
    BufferedCopy(inStream, outStream, inStream.GetSize());
}
```

wxFFileInputStream和wxFFileOutputStream跟wxFileInputStream和 wxFileOutputStream的用法几乎完全一样, 不同之处在于前者是基于wxFFile类而不是wxFile类的. 因此它们的初始化方法也不同,相应的,前者可以使用FILE指针或wxFFile对象来初始化.文件结束处理相应的有所不同: wxFileInputStream在最后一个字节被读取的时候报告wxSTREAM_EOF, 而wxFFileInputStream则在最后一个字节以后进行读操作的时候返回wxSTREAM_EOF.

内存和字符串流

wxMemoryInputStream和wxMemoryOutputStream使用内部缓冲区来处理流数据.默认的构造函数都采用 char*类型的缓冲区指针和缓冲区大小作为参数.如果没有这些参数,则表明要求该类的事例自己进行动态缓冲区管理.我们很快会看到一个相关的例子.

wxStringInputStream则采用一个wxString引用作为构造参数来进行数据读取. wxStringOutputStream采用一个开选的wxString指针作为参数来进行数据写操作;如果构造参数没有指示wxString指针,则将构造一个内部的wxString对象,这个对象可以通过GetString函数来访问.

读写数据类型

到目前为止,我们描述的流类型都处理的是原始的字节流数据.在实际的应用程序中,这些字节流必须被赋予特定的函数.为了帮助你实现这一点,你可以使用下面四个类来以一个更高的层级处理数据.这四个类分别是:wxTextInputStream, wxTextOutputStream, wxDataInputStream和wxDataOutputStream.这些类通过别的流类型类构造,它们提供了操作更高级的C++数据类型的方法.

wxTextInputStream从一段人类可读的文本中获取数据.如果你使用的构造类为文件相关类,你需要自己进行文件是否读完的判断.即使这样,读到空数据项(长度为0的字符串或者数字0)还是无法避免,因为很多文本文件都用空白符(比如换行符)来结束.下面的例子演示了怎样使用由wxFileInputStream构造的wxTextInputStream:

```
wxFileInputStream input( wxT("mytext.txt") );
wxTextInputStream text( input );
wxUint8 i1;
float f2;
wxString line;
text &gt;&gt; i1;       // 读一个8bit整数.
text &gt;&gt; i1 &gt;&gt; f2; // 先读一个8bit整数再读一个浮点数.
text &gt;&gt; line;     // 读一行文本
```

wxTextOutputStream则将文本数据写至输出流,换行符自动使用当前平台的换行符.下面的例子演示了将文本数据输出到标准错误输出流:

```
#include "wx/wfstream.h"
#include "wx/txtstrm.h"
wxFFileOutputStream output( stderr );
wxTextOutputStream cout( output );
cout &lt;&lt; wxT("This is a text line") &lt;&lt; endl;
cout &lt;&lt; 1234;
cout &lt;&lt; 1.23456;
```

wxDataInputStream和wxDataOutputStream使用发放类似,但是它使用二进制的方法处理数据.数据使用可移植的方式存储因此能够作到平台无关.下面的例子分别演示了以这种方式从数据文件读取以及写入数据文件.

```
#include "wx/wfstream.h"
#include "wx/datstrm.h"
wxFileInputStream input( wxT("mytext.dat") );
wxDataInputStream store( input );
wxUint8 i1;
float f2;
wxString line;
store &gt;&gt; i1;       // 读取一个8bit整数
store &gt;&gt; i1 &gt;&gt; f2; // 读取一个8bit整数,然后读取一个浮点数.
store &gt;&gt; line;     // 读取一行文本

#include "wx/wfstream.h"
#include "wx/datstrm.h"
wxFileOutputStream output(wxT("mytext.dat") );
wxDataOutputStream store( output );
store &lt;&lt; 2 &lt;&lt; 8 &lt;&lt; 1.2;
store &lt;&lt; wxT("This is a text line") ;
```

Socket流

wxSocketOutputStream和wxSocketInputStream是通过wxSocket对象构造的,详情参见第18章.

过滤器流对象

wxFilterInputStream和wxFilterOutputStream是过滤器流对象的基类,过滤器流对象是一种特殊的流对象,它用来将过滤后的数据输入到其它的流对象.wxZlibInputStream是一个典型的过滤器流对象.如果你在其构造函数中指定一个文件源为一个zlib格式的压缩文件的文件流对象,你可

以直接从wxZlibInputStream中读取数据而不需要关心解压缩的机制.类似的,你可以使用一个wxFileOutputStream来构造一个wxZlibOutputStream对象,如果你将数据写入wxZlibOutputStream对象,压缩后的数据将被写入对应的文件中.

下面的例子演示了怎样将一段文本压缩以后存放入另外一个缓冲区中:

```
#include "wx/mstream.h"
#include "wx/zstream.h"
const char* buf =
    "01234567890123456789012345678901234567890123456789";
// 创建一个写入wxMemoryOutputStream对象的wxZlibOutputStream类
wxMemoryOutputStream memStreamOut;
wxZlibOutputStream zStreamOut(memStreamOut);
// 压缩buf以后写入wxMemoryOutputStream
zStreamOut.Write(buf, strlen(buf));
// 获取写入的大小
int sz = memStreamOut.GetSize();
// 分配合适大小的缓冲区
// 拷贝数据
unsigned char* data = new unsigned char[sz];
memStreamOut.CopyTo(data, sz);
```

Zip流对象

wxZipInputStream是一个更复杂一点的流对象,因为它是以文档的方式而不是线性的二进制数据的方式工作的.事实上,文档是通过另外的类wxArchiveClassFactory和wxArchiveEntry来处理的,但是你可以不用关心这些细节.要使用 wxZipInputStream类,你可以有两种构造方法,一种是直接使用一个指向zip文件的文件流对象,另外一种方法则是通过一个zip文件路径和一个zip文件中文档的路径来指定一个zip数据流.下面的例子演示了这两种方法:

```
#include "wx/wfstream.h"
#include "wx/zipstrm.h"
#include "wx/txtstrm.h"
// 方法一: 以两步方式创建zip输入流.
wxZipEntry* entry;
wxFFileInputStream in(wxT("test.zip"));
wxZipInputStream zip(in);
wxTextInputStream txt(zip);
wxString data;
while (entry = zip.GetNextEntry())
{
    wxString name = entry->GetName();    // 访问元数据
    txt &gt;&gt; data;                          // 访问数据
    delete entry;
}
// 方法二: 直接指定源文档路径和内部文件路径.
wxZipInputStream in(wxT("test.zip"), wxT("text.txt"));
wxTextInputStream txt(zip);
wxString data;
txt &gt;&gt; data;                              // 访问数据
```

wxZipOutputStream用来写zip压缩文件. PutNextEntry或PutNextDirEntry函数用来在压缩文件中创建一个新的文件(目录),然后就可以写相应的数据了.例如:

```
#include "wx/wfstream.h"
#include "wx/zipstrm.h"
#include "wx/txtstrm.h"
wxFFileOutputStream out(wxT("test.zip"));
wxZipOutputStream zip(out);
wxTextOutputStream txt(zip);
zip.PutNextEntry(wxT("entry1.txt"));
txt &lt;&lt; wxT("Some text for entry1\n");
zip.PutNextEntry(wxT("entry2.txt"));
txt &lt;&lt; wxT("Some text for entry2\n");
```

虚拟文件系统

wxWidgets提供了一套虚拟文件系统机制,让你的应用程序可以象使用普通文件那样使用包括zip文件中的文件,内存文件以及HTTP或FTP协议这样的特殊数据.不过,这种虚拟文件机制通常是只读的,意味着你不可以修改其中的内容.wxWidgets提供的wxHtmlWindow类(用于提供wxWidgets内部的HTML帮助文件的显示)和wxWidgets的XRC资源文件机制都可以识别虚拟文件系统路径格式.虚拟文件系统的使用比起前面介绍的zip文件流要简单,但是后者可以更改zip文档的内容.除了内部都使用了流机制以外,这两者其实没有任何其它的联系.

各种不同的虚拟文件系统类都继承自wxFileSystemHandler类,要在应用程序中使用某个特定实现,需要在程序的某个地方 (通常是OnInit函数中)调用wxFileSystem::AddHandler函数.使用虚拟文件系统通常只需要使用那些定义在 wxFileSystem对象中的函数,但是有些虚拟文件系统的实现也提供了直接给用于使用的函数,比如wxMemoryFSHandler's的 AddFile和RemoveFile函数.

在我们介绍怎样通过C++函数访问虚拟文件系统之前,我们先看看怎样在wxWidgets提供的其它子系统中使用虚拟文件系统.下面的例子演示了怎样在用于在wxHtmlWindow中显示的HTML文件中使用指定虚拟文件系统中的路径:

```
<img src="file:myapp.bin#zip:images/logo.png">
```

"#"号前面的部分是文件名,后面的部分则是虚拟文件系统类型以及文件在虚拟文件系统中的路径.

类似的,我们也可以在XRC资源文件中使用虚拟文件系统:

```
<object class="wxBitmapButton">
    <bitmap>file:myapp.bin#zip:images/fuzzy.gif</bitmap>
</object>
```

在上面的这些用法中,操作虚拟文件系统的代码被隐藏在wxHtmlWindow和XRC系统的实现中.如果你希望直接使用虚拟文件系统, 通常你需要通过wxFileSystem和wxFSFile类.下面的代码演示了怎样从虚拟文件系统中加载一幅图片,当应用程序初始化的时候,增加一个wxZipFSHandler类型的虚拟文件系统处理器.然后创建一个wxFileSystem的实例,这个实例可以是临时使用也可以存在于整个应用程序的生命周期,这个实例用来从zip文件myapp.bin中获

取logo.png图片.wxFSFile对象用于返回这个文件对应的数据流,然后通过流的方式创建wxImage对象.在这个对象被转换成wxBitmap格式以后,wxFSFile和wxFileSystem对象就可以被释放了.

```
#include "wx/fs_zip.h"
#include "wx/filesys.h"
#include "wx/wfstream.h"
// 这一行代码只应该被执行依次,最好是在应用程序初始化的时候
wxFileSystem::AddHandler(new wxZipFSHandler);
wxFileSystem* fileSystem = new wxFileSystem;
wxString archive = wxT("file:///c:/myapp/myapp.bin");
wxString filename = wxT("images/logo.png");
wxFSFile* file = fileSystem->OpenFile(
              archive + wxString(wxT("#zip:")) + filename);
if (file)
{
    wxInputStream* stream = file->GetStream();
    wxImage image(* stream, bitmapType);
    wxBitmap bitmap = wxBitmap(image);
    delete file;
}
delete fileSystem;
```

注意要使用wxFileSystem:: OpenFile函数,其参数必须是一个URL而不能是一个绝对路径,其格式应该为"file:/<主机名>//<文件名>", 如果主机名为空,则使用三个"/"符号.你可以使用wxFileSystem::FileNameToURL函数获取某个文件对应的URL,也可以用wxFileSystem::URLToFileName函数将某个URL转换成对应的文件名.

下面的例子演示了怎样获取虚拟文件系统中获取一个文本文件的内容,并将其存放在某个wxString对象中,所需参数为zip文件路径以及zip文件中的虚拟文件的路径:

```
// 从zip文件中加载一个文本文件
bool LoadTextResource(wxString& text, const wxString& archive,
                      const wxString& filename)
{
    wxString archiveURL(wxFileSystem::FileNameToURL(archive));
    wxFileSystem* fileSystem = new wxFileSystem;
    wxFSFile* file = fileSystem->OpenFile(
                 archiveURL + wxString(wxT("#zip:")) + filename);
    if (file)
    {
        wxInputStream* stream = file->GetStream();
        size_t sz = stream->GetSize();
        char* buf = new char[sz + 1];
        stream->Read((void*) buf, sz);
        buf[sz] = 0;
        text = wxString::FromAscii(buf);
        delete[] buf;
        delete file;
        delete fileSystem;
        return true;
    }
    else
        return false;
}
```

wxMemoryFSHandler允许你将数据保存在内存中并且通过内存协议在虚拟文件系统中使用它.显然在内存中存放大量数据并不是一个值得推荐的作法,不过有时候却可以给应用程序提供一定程序的灵活性,比如你正在使用只读的文件系统或者说使用磁盘文件存在性能上的问题的时候.在 DialogBlocks软件中,如果用户提供的自定义图标文件还不具备,DialogBlocks仍然可以通过下面的XRC文件显示一个内存中的图片, 这个图片并不存在于任何的物理磁盘中.

```
<object class="wxBitmapButton">
    <bitmap&gt;memory:default.png</bitmap>
</object>
```

wxMemoryFSHandler的AddFile函数可以使用的参数包括一个虚拟文件名和一个wxImage, wxBitmap, wxString或void*数据. 如果你不再使用某个内存虚拟文件了,可以通过RemoveFile函数将其删除.如下所示:

```
#include "wx/fs_mem.h"
#include "wx/filesys.h"
#include "wx/wfstream.h"
#include "csquery.xpm"
wxFileSystem::AddHandler(new wxMemoryFSHandler);
wxBitmap bitmap(csquery_xpm);
wxMemoryFSHandler::AddFile(wxT("csquery.xpm"), bitmap,
                           wxBITMAP_TYPE_XPM);
...
wxMemoryFSHandler::RemoveFile(wxT("csquery.xpm"));
```

wxWidgets支持的第三种虚拟文件系统是wxInternetFSHandler,它支持FTP和HTTP协议.

第十四章小结

本章介绍了怎样使用wxWidgets提供的跨平台的文件和流操作相关的类.还介绍了wxWidgets中的虚拟文件系统机制,它可以让你很方便的访问位于zip文件,内存或者Internet上的文件.

在下一章里,我们将介绍一些你的最终用户可能不会直接看到但是却仍然非常重要的话题:内存管理,调试和错误检测.

第十五章内存管理,调试和错误处理

找到应用程序中的错误是任何开发过程中一个最基本的部分,虽然,也许它并不是那么有诱惑力.本章用来介绍wxWidgets提供的用来检测内存问题的机制.同时也涉及了一些构建更可靠和更容易调试的应用程序的内容,比如怎样构建具有自防御功能的程序.我们还将解释什么时候应该在堆上创建对象,什么时候则应该在栈上创建对象,以及怎样使用运行期类型信息机制,模块机制以及wxWidgets提供的C++异常支持等.最后,我们会提供一些通用的调试方法提示.

15.1 内存管理基础

和大多数的C++程序一样,你可以在栈上创建对象,也可以在堆上通过new来创建对象.前者的生命周期仅限于其作用范围,当代码离开其作用范围时,对象的析构函数被调用,对象被释放.而堆上的对象则相反,它将一直存在除非你使用delete操作释放对象或者整个应用程序退出.

创建和释放窗口对象

一个通用的规则是:想frame或者按钮这样的窗口对象总是应该在堆上使用new方法创建.因为窗口对象通常有一个不确定的生命周期.或者说,窗口对象通常要等用户决定什么时候关闭并且释放自己.注意wxWidgets会自动释放子窗口,因此你只需要调用顶层窗口的Destroy函数,而不需要依次调用窗口中其它控件的Destroy函数.类似的,在被释放的时候,frame窗口也会自动释放它的所有的子窗口.不过如果你创建了一个作为 frame窗口子窗口的顶层窗口(比如另外一个frame),这个子窗口将不会被自动释放.这又有一个例外,在MDI(多文档界面)中,子文档窗口将被自动释放,因为它们其实是主文档窗口的一部分,是不应该独立存在的.

你可以在栈上创建对话框窗口,但是它必须是模式对话框.换句话说,你必须使用ShowModal函数使其进入一个事件循环,所有的和用户的交互都将在这个事件循环中被处理,并且这些交互都发生在对话框离开其作用域从而被释放之前.

释放frame窗口和对话框窗口的方法也许让你感到困惑,我们进一步解释一下.要释放frame窗口或者非模式对话框,应用程序应该调用 Destroy函数,这个函数在相关对话框的事件队列变空以后才会释放窗口,否则可能将事件发送到不存在的窗口导致程序异常.然而对于模式对话框来说,应该首先调用EndModal函数退出窗口的事件循环.这个函数通常是在对话框的事件处理函数中被调用的,比如默认的OK按钮事件处理函数.在事件处理函数中只能调用EndModal而不能调用Destroy,因为模式对话框很可能是在栈上创建的,如果在事件处理函数中调用Destroy,很可能导致这个对话框被重复释放(一次是在事件处理函数中,一次是在栈变量退出其作用域时)导致应用程序表现异常.因此,正确的处理顺序是:当用户点击关闭按钮时,产生wxEVT_CLOSE_WINDOW事件,这个事件的处理函数调用EndModal函数(而不是Destroy函数).而对话框则可以在其退出作用域的时候被自动释放,这也是所有wxWidgets提供的标准对话框采用的方式.这种方式的另外一个好处在于,当调用EndModal函数退出事件循环以后, 你还可以访问对话框变量的公共成员以便获取相应的数据.当然你也可以设计直接在其事件处理函数中释放自己的对话框,不过对于这种对话框,你就不能够在栈上创建它并且也不可以在它被关闭以后还可以访问它的公共成员了.

下面演示的两种使用wxMessageDialog的方法都是允许的:

```
// 1) 在栈上创建模式对话框,没有显式的释放函数
wxMessageDialog dialog(NULL, _("Press OK"), _("App"), wxOK|wxCANCEL);
if (dialog.ShowModal() == wxID_OK)
{
    // 2) 创建在堆上的模式对话框,必须使用Destroy显式删除
    wxMessageDialog* dialog = new wxMessageDialog(NULL,
        _("Thank you! "), _("App"), wxOK);
    dialog->ShowModal();
    dialog->Destroy();
}
```

非模式对话框和frame窗口通常需要在关闭的时候释放自己,关闭动作可以通过各种方式产生.它们不能在栈上创建,因为它们通常会很快离开自己的作用范围,将被立即释放.

如果你使用了指向窗口对象的指针,需要确定在窗口被释放以后,指针被置为NULL,这可以在窗口的析构函数或者关闭事件处理函数中完成,如下所示:

```
void MyFindReplaceDialog::OnCloseWindow(wxCloseEvent& event)
{
    wxGetApp().SetFindReplaceDialog(NULL);
    Destroy();
}
```

创建和复制绘画对象

绘画过程中使用的对象,比如wxBrush, wxPen, wxColour, wxBitmap和wxImage等,既可以在堆上创建也可以在栈上创建.这些对象在绘画函数中使用时通常在栈上创建(局部变量).这些对象内部使用引用记数机制,因此直接使用对象(而不是对象指针)的系统开销也是非常小的.这种机制的具体作法是:在进行赋值或复制操作的时候,只复制对象内部数据的一个引用而不是整个的内部数据.当然这种机制有时候也会导致一些奇怪的问题,比如某个对象的修改导致所有引用该对象的对象的修改.为了避免这种情况,在需要的时候,你可以通过在构造函数中传递具体的参数来构造一个全新的对象.下面各自举一些例子:

```
// 引用记数
wxBitmap newBitmap = oldBitmap;
// 全新拷贝
wxBitmap newBitmap = oldBitmap.GetSubBitmap(
    wxRect(0, 0, oldBitmap.GetWidth(), oldBitmap.GetHeight()));
// 引用记数
wxFont newFont = oldFont;
// 全新拷贝
wxFont newFont(oldFont.GetPointSize(), oldFont.GetFamily(),
    oldFont.GetStyle(), oldFont.GetWeight(),
    oldFont.GetUnderlined(), oldFont.GetFaceName());
```

初始化你的应用程序中的对象

因为你的自定义对象可能先于wxWidgets自己的初始化而被创建,有可能在你的自定义对象初始化的时候,wxWidgets的内部数据包括颜色数据库,字体数据库等都还没有被初始化,因此,不要在你的自定义应用程序类的构造函数中使用wxwidgets的预定义值初始化你的对象,如果你需要使用这些值,可以在OnInit函数中进行.举例如下:

```
MyApp::MyApp()
{
    // 不要在这个时候作初始化
    // m_font = *wxNORMAL_FONT;
}
bool MyApp::OnInit()
{
    m_font = *wxNORMAL_FONT;
    ...
    return true;
}
```

在应用程序退出时执行清理

你可以在重载的wxApp::OnExit函数中执行大多数的清理工作,这个函数是在所有的窗口都已经被释放,wxWidgets正准备释放自己的内部数据的时候被调用的.不过有些清理工作还是需要在主窗口的关闭事件处理函数中执行,比如关闭那些可能往主窗口发送数据的进程.

15.2 检测内存泄漏和其它错误

在理想状态下,当你的应用程序退出时,所有的对象都应该被你的应用程序或者wxWidgets本身释放,不会有残留的内存需要操作系统自己去释放.不用关心内存的释放听上去似乎非常的诱人,但是,我们还是要说,你应该自己控制所有对象的释放,而不要把它留给操作系统,因为通常,内存泄漏都是一些其它重大问题的前兆,它可能导致你的系统在一段时间内出现严重的内存问题.而当你已经转而关注程序的其它方面的时候,会过头来研究哪里出现了泄漏是一件非常令人沮丧的事情,因此,让你的程序尽量保持对内存泄漏的零容忍度不是一个坏事.

那么,怎样才可能让你的应用程序拥有这种检测内存泄漏的能力呢?各种各样的第三方工具提供了这种能力甚至更多其它的能力,而且, wxWidgets也提供了一个自己的简单的内建的内存检测器.如果你想使用它,在windows平台上,你需要打开setup.h中的几个开关,而在 linux平台上,configure程序需要一些特殊的开关:

在windows平台上,你需要:

```
#define wxUSE_DEBUG_CONTEXT 1
#define wxUSE_MEMORY_TRACING 1
#define wxUSE_GLOBAL_MEMORY_OPERATORS 1
#define wxUSE_DEBUG_NEW_ALWAYS 1
```

而对于configure程序,需要传递这样的参数:

```
--enable-debug --enable-mem_tracing --enable-debug_cntxt
```

另外,使用这个系统还有一个限制:到作者停笔之前,它还不支持MinGW或Cygwin编译器,并且如果你的代码中使用了STL或者使用CodeWarrior编译器,你将不能使用wxUSE_DEBUG_NEW_ALWAYS选项.

如果打开了wxUSE*DEBUGNEWALWAYS*选项,那么所有使用*new*操作分配对象的地方都将被*new (_TFILE,__LINE*)代替,后者已经被重新使用调试版本的定制内存分配和释放机制进行实现.如果想不重新定义new而显式使用调试版本的new过程,你可以在使用new的地方用WXDEBUG_NEW代替.

最简单的使用这个系统的方法是:什么都不做,就只运行你的程序,作该做的事情,然后退出应用程序,然后检查是否有任何内存泄漏被报告.下面的内存泄漏报告的一个例子:

```
There were memory leaks.

- Memory dump -
.\memcheck.cpp(89): wxBrush at 0xBE44B8, size 12
..\..\src\msw\brush.cpp(233): non-object data at 0xBE55A8, size 44
.\memcheck.cpp(90): wxBitmap at 0xBE5088, size 12
..\..\src\msw\bitmap.cpp(524): non-object data at 0xBE6FB8, size 52
.\memcheck.cpp(93): non-object data at 0xBB8410, size 1000
.\memcheck.cpp(95): non-object data at 0xBE6F58, size 4
.\memcheck.cpp(98): non-object data at 0xBE6EF8, size 8

- Memory statistics -
1 objects of class wxBitmap, total size 12
5 objects of class nonobject, total size 1108
1 objects of class wxBrush, total size 12

Number of object items: 2
Number of non-object items: 5
Total allocated size: 1132
```

上面的例子显示有一个wxBrush对象和一个wxBitmap对象被分配了但是没有被释放,还有一些其它的未知的对象没有被释放,因为它们没有提供wxWidgets的运行期类型信息,因此无法确定对象的类型.在某些集成开发环境中,你可以通过双击报告行显示相应的代码中分配这个对象的位置,这通常是检查内存泄漏问题一个很好的开始.当然,为了最好的报告效果,最好给任何继承自wxObject都提供运行期类型信息,方法是,在类声明的部分增加DECLARE_CLASS(class)宏,在类实现的地方增加IMPLEMENT_CLASS(class, parentClass)宏.

这个内存检测系统还试图检测那些内存越界或重复释放的错误.在分配内存的时候,它自动在已经分配的内存块上设置一个"good"标记, 在释放的时候将会检查这个标记,如果释放的内存块没有这个标记,则报告一个内存释放错误.这将帮助你发现那些隐藏的,也许在多次运行以后才会导致系统异常的内存错误.

wxDebugContext类的一些静态函数也很有用处,你可以通过PrintClasses函数获取当前系统中分配的对象的列表, PrintStatistics函数可以打印出当前系统中的已知类型对象和未知类型对象的个数.使用SetCheckpoint函数,你可以告诉 wxDebugContext忽略这个函数调用以前的内存分配动作,而仅关注这以后的内存分配.详情请参考samples/memcheck例子或者wxDebugContext的相关手册.

除了wxWidgets内建的基本系统,你还可以使用别的商业的或者免费的内存检测软件,商业软件比如:BoundsChecker, Purify或AQtime等.自由软件比如:StackWalker,ValGrind,Electric Fence或来自Fluid Studios的MMGR等.如果你使用的是Visual C++,wxWidgets使用的是编译器内置的内存检测机制,它不会给出类的名字但是会给出行号.最好是打开wxUSE_DEBUG_NEW_ALWAYS选项,因为它将重定义new过程,除非打开这个选项导致别的第三方的库出现兼容方面的问题.

15.3 构建自防御的程序

通常一个缺陷在导致其产生的逻辑错误产生很久以后才会表现出来.如果一个异常的或者不正确的值没有被及时检测到,那么通常在应用程序又执行了几千行代码以后,应用程序才会崩溃或者作出令人费解的表现.为解决这样的问题你可能要花上很长的时间.但是,如果你在你的代码中增加一些普通的检查,来检测函数的某个地方出现了错误的值,那么你的代码将最终变得非常强壮,你将可能避免你和你的用户陷入到大麻烦中去.这就是所谓的自防御程序.你的代码可以防止自己被别的代码或者自己内部的某些错误逻辑搞的一团糟.因为大多数的这些检测在正式发布版中都将被移除,因此,它们占用的系统开销几乎可以忽略不计.

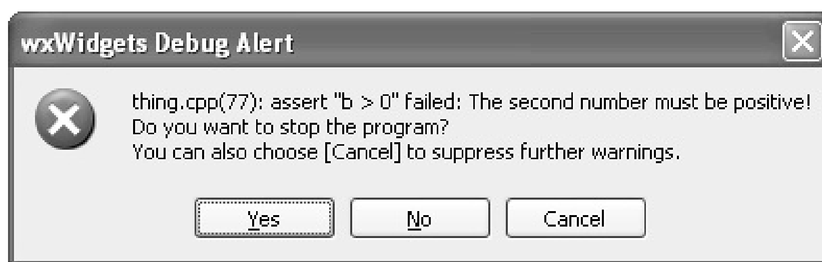
正如你期待的那样,wxWidgets在其内部使用了大量的错误检测代码,你可以在你的代码中直接使用这些宏.这些宏主要分为三类,每一类都有多种形式:第一类是wxASSERT,如果它的参数不等于True,它将显示一个错误信息.这种检测仅存在于调试版本中.wxFAIL则将使用产生一个错误信息,其作用相当于wxASSERT(false).它也仅会存在于调试版本中.wxCHECK则判断条件是否成立,如果不成立则返回某个值并显示错误信息.和前面两种不同的是,在正式版本中,wxCHECK的代码仍然有效,但是将不显示任何消息.

下面的例子演示了怎样使用这些宏:

```
// 两个正数相加
int AddPositive(int a, int b)
{
    // 检查是否为一个正数
    wxASSERT(a &gt; 0);
    // 检查是否位一个正数,显示定制的消息
    wxASSERT_MSG(b &gt; 0, wxT("The second number must be positive!"));
    int c = a + b;
    // 如果相加的结果不为正数,显示一个错误的消息并且返回-1.
    wxCHECK_MSG(c &gt; 0, -1, wxT("Result must be positive!"));
    return c;
}
```

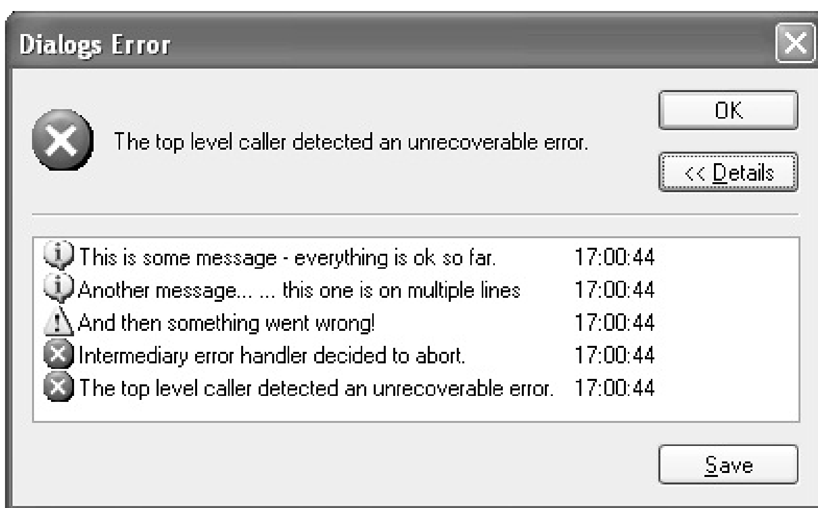
你还可以使用wxCHECK2和wxCHECK2_MSG宏,这两个宏在条件不满足的时候可以执行任意的操作而不仅仅是设置一个返回值. 而wxCHECK_RET则可以被用在没有返回值(void)的函数中.另外一些不太常用的宏包括wxCOMPILE_TIME_ASSERT和 wxASSERT_MIN_BITSIZE等,请参考wxWidgets的相关手册.

下图演示了一个断言不满足时显示消息的对话框的样子,在这个对话框上有三个按钮,Yes按钮停止当前程序的运行,No按钮则忽略这个断言失败,而Cancel按钮则表示以后的断言失败都不需要显示.如果程序正运行在某个调试器中,则程序终止运行将返回到调试器中,你可以打印出当前的函数调用堆栈,进而可以知道断言失败的准确位置以及当时各个参数的值.


wxWidgets Debug Alert
thing.cpp(77): assert "b > 0" failed: The second number must be positive!
Do you want to stop the program?
You can also choose [Cancel] to suppress further warnings.
Yes
No
Cancel

15.4 错误报告

有时候你需要在控制台或者一个对话框中显示一条消息以帮助调试或者用来提示那些不能被你的代码正常处理的行为.wxWidgets提供了很多用于记录错误的函数,这些函数工作方式各不相同,你可以使用它们来进行运行情况的报告和记录.比如,当你正在分配一个很大的图片时,可能由于这个图片太大了,系统无法分配足够的资源,系统将使用wxLogError函数显示一个对话框来报告这个错误(如下图所示).又或者说,你想将某个参数的值打印在调试窗口中以便于调试,你可以使用wxLogDebug函数.究竟这些错误信息或者调试信息是显示在终端上,对话框中还是别的什么地方,取决于你所使用的函数,以及当前激活的wxLog目标对象,我们将在稍后的部分描述相关内容.



所有的这种记录函数都拥有类似于printf或vprintf的语法,也就是:第一个参数是格式化文本参数,后面是不定类型的变量或者一组指向变量的指针,如下所示:

```
wxString name(wxT("Calculation"));
int nGoes = 3;
wxLogError(wxT("%s does not compute! You have %d more goes."),
           name.c_str(), nGoes);
```

下面我们逐个描述一下这些函数:

wxLogError函数用来显示那些必须显示给用户的错误消息.其默认行为是弹出一个对话框来通知用户相应的错误.那为什么不直接使用 wxMessageBox呢?原因是,首先wxLogError提示的错误消息是可以通过创建一个wxLogNull的Log目标来屏蔽让其不显示出来的,而且这些消息也是排在系统队列中,在系统空闲的时候显示的.因此如果有一系列的错误同时出现,它们将显示在同一个对话框中,而如果使用 wxMessageBox,你的用户可能不得不不停的点击OK按钮.

wxLogFatalError和wxLogError类似,不过除了显示错误消息,它还使用标准的系统调用abort,以错误码3结束整个程序的运行.和其它类似的函数不同,这个函数显示的消息不能通过设置空的打印目标的方法来屏蔽.

wxLogWarning也和wxLogError类似,不过显示的信息将作为警告而不是错误.

wxLogMessage则用来显示所有正常的,信息类型的消息,默认也是显示在对话框中.

wxLogVerbose则用来显示那些冗长的详细信息.通常情况下,这种信息是不显示的,但是如果用户想显示它以便了解程序运行的更详细的情况,可以通过使用wxLog::SetVerbose函数改变这种默认的行为.

wxLogStatus则用来显示状态条消息,如果当前的frame窗口拥有一个状态条,那么这个消息将显示在那里.

wxLogSysError通常主要被wxWidgets自己使用,它用来报告那些系统错误,同时会显示由errno或者GetLastError(依平台的不同)指示的错误码和错误消息.它的另外一种形式允许你的第一个参数的位置显式的指示系统错误码.

wxLogDebug用来显式调试信息.这些信息只在调试版本中(定义了**WXDEBUG**宏)才会出现,在正式版本中将被移除.在 windows平台上,只有当程序在一个调试器中运行或者使用第三方工具比如来自http://www.sysinternals.com的 DebugView工具运行的时候才会显示出来.

wxLogTrace和wxLogDebug的功能几乎完全一样,也是只在调试模式才会显示信息.之所以有这个函数,是为了提供一个和普通的调试模式不同的级别以便区分普通的调试信息和用于跟踪的调试信息.它的另外一种形式允许你指定一个掩码,通过wxLog:: AddTraceMask函数设置了掩码以后,只有掩码符合的跟踪消息才会被显示出来以实现跟踪消息的过滤.比如在wxWidgets内部使用了 mousecapture掩码.如果你设置了这个掩码,在鼠标移动的时候你将看到跟踪信息.

```
void wxWindowBase::CaptureMouse()
{
    wxLogTrace(wxT("mousecapture"), wxT("CaptureMouse(%p) "), this);
...
}
void MyApp::OnInit()
{
    // Add mousecapture to the list of trace masks
    wxLog::AddTraceMask(wxT("mousecapture"));
    ...
}
```

你可能会疑惑,为什么不直接使用C的标准输入输出函数或者C++的流呢?简短的回答是,它们都是很不错的机制,但是并不一定适用于wxWidgets.wxLog机制主要有以下三个优点:

首先,wxLog是可移植的.常用的printf语句或者C++的cout流和cerr流在unix系统下工作是没有问题的,但是在 windows系统中,对于图形化界面的应用程序,这些函数或流可能不能正常显示需要的内容.因此,你可以使用wxLogMessage作为printf 的一个简单的替代品.

你也可以通过下面的方法将所有的log信息转向标准的cout流:

```
wxLog *logger=new wxLogStream(&cout);
wxLog::SetActiveTarget(logger);
```

另外,将发送往cout的输出重定向到一个wxTextCtrl控件也是可行的,这需要使用到wxStreamToTextRedirector类.

其次:wxLog更灵活.使用wxLog机制的输出可以被分情况重定向或者隐藏,比如只显示错误消息和告警消息,忽略所有正常的信息.而如果使用标准的函数或流,这是不可能的或者说是很难作到的.

最后:wxLog机制也是更完善的机制.通常,当有错误发生的时候,应该给用户显示一些信息.让我们来举一个简单的例子,假如你正在进行写文件操作,这时候发生了磁盘空间不足的情况,这种错误是被wxWidgets内部(wxFile::Write)处理的,因此,调用这个函数只能知道写动作发生了异常,至于是什么类型的异常则很难得到,如果在这种情况下使用wxLogError函数,正确的错误码和相应的错误信息都将显示给用户.

现在我们来描述以下wxWidgets的Log机制是怎样工作的,以便你处理那些默认没有提供的行为.

wxWidgets有一个log目标的概念:它其实就是一个wxLog的派生类.它需要实现wxLog定义的那些虚函数,这些函数将在相应的Log函数被调用的时候使用.任何时候都只有一个log目标是活动的.log目标通常的使用方法就是调用wxLog:: SetActiveTarget来安装这个目标,安装以后的目标将在相应的log函数被调用的时候自动使用.

要创建一个自定义的log目标,你只需要创建一个wxLog的派生类,并实现其虚函数DoLogString和(或)DoLog.如果你对wxWidgets默认的增加时间戳和信息类型的格式化方法感到满意,只是想更改信息的目的地,那么实现DoLogString函数就足够了,而重载 DoLog函数则使得你可以任意的定制输出信息的格式,不过同时你也需要自己区分信息的各种类型.你可以参考src/common/log.cpp文件看看wxWidgets是怎么作到这一点的.

wxWidgets自己实现了几个wxLog的派生类,你也可以读一读它们的代码,这对你创建自定义的log目标也是有好处的.这些预定义的log目标包括:

wxLogStderr将所有的信息输出到FILE作为参数的文件中,如果*FILE*为空,则输出到标准错误输出.

wxLogStream和wxLogStderr功能相同,不过它使用标准C++的ostream类和cerr流来代替FILE*和stderr.

wxLogGui则是wxWidgets所有wxWidgets程序默认使用的log目标,依平台的不同它实现了不同的输出处理.

wxLogWindow则提供了一个类似"跟踪终端"之类的窗口,这个窗口将显示所有的输出信息,同时这些信息也将显示在之前的log目标上.这个跟踪终端窗口提供了清除信息,关闭窗口以及将所有信息保存到文件中的功能.

wxLogNull则被用来临时阻止某些错误信息的输出,比如你打开不存在的文件的时候将显示一个错误信息,有时候你不希望显示这个信息,可以在栈上创建一个wxLogNull变量,在这个变量的作用域范围内,没有任何错误信息将被显示,而离开了其作用域,则所有的信息又可以正常显示了.

```
wxFile file;
// wxFile.Open()在打开一个不存在的文件时通常会显示错误信息,但是在这里我们不想看到这个信息.
{
    wxLogNull logNo;
    if ( !file.Open("bar") )
      ... process error ourselves ...
} // wxLogNull的析构函数被调用,旧的log目标被恢复.
wxLogMessage("..."); // 可以被显示
```

有时候你也许希望将信息输出到多个地方,比如,你可以希望所有的信息在正常显示的同时被保存在某个文件中,这时候你可以使用wxLogChain和wxLogPassThrough,如下所示:

```
// 这将隐式的设置当前log目标
wxLogChain *logChain = new wxLogChain(new wxLogStderr);
// 所有的输出将被同时显示在stderr和通常的地方
// 不要直接删除logChain指针,这会导致当前活动log目标为一个不确定的指针.
// 应该使用SetActiveTarget.
delete wxLog::SetActiveTarget(new wxLogGui);
```

wxMessageOutput VS wxLog

有时候,使用wxLog不太合适,这主要是因为wxLog对输出的信息作了过多的处理,并且会等待空闲的时候才会显示这些信息.而 wxMessageOutput和它的派生类则可以作为你的底层printf的替代品,来在GUI和命令行程序中使用.你可以象使用printf函数那样使用wxMessageOutput::Printf函数,比如,如果你想把信息打印在标准错误输入:

```
#include "wx/msgout.h"
wxMessageOutputStderr err;
err.Printf(wxT("Error in app %s.\n"), appName.c_str());
```

wxMessageOutputDebug将信息显示在调试器终端或者是标准错误输出中,这主要看程序是以什么方式运行的,和 wxLogDebug不同,wxMessageOutputDebug输出的信息在正式版本中将不会被移除.GUI应用程序还可以使用 wxMessageOutputMessageBox来即时显示消息,而不比象wxLog那样需要搜集(其它Log信息)和等待(系统空闲),同样的还存在一个wxMessageOutputLog类,它将消息输出到wxLogMessage.

和wxLog类似,wxMessageOutput也有一个当前的目标,这个目标可以通过wxMessageOutput::Set设置,通过wxMessageOutput::Get获取.默认的目标是系统初始化的时候由wxWidgets设置的,在命令行程序中使用的是 wxMessageOutputStderr,在GUI程序中使用的是wxMessageOutputMessageBox.wxWidgets内部经常使用这个对象,比如在wxCmdLineParser类中,使用了下面的代码:

```
wxMessageOutput* msgOut = wxMessageOutput::Get();
if ( msgOut )
{
    wxString usage = GetUsageString();
    msgOut->Printf( wxT("%s%s"), usage.c_str(), errorMsg.c_str() );
}
else
{
    wxFAIL_MSG( _T("no wxMessageOutput object?") );
}
```

15.5 提供运行期类型信息

提供运行期类型信息

和大多数编程框架一样,wxWidgets提供了比标准C++更多的运行期类型信息.这对于在运行期依对象类型决定行为,或者在上节提到的错误报告中都是很有用处的.并且它还允许你通过对象名来创建一个那种类型的变量.需要注意的是:只有派生自wxObject的类才可以使用wxWidgets的RTTI (运行期类型信息).

如果你不需要动态创建功能,你需要在类声明中使用DECLARE*CLASS(class)*宏,而在类实现中使用 *IMPLEMENT_CLASS(class, baseClass)*宏,反之,则需要使用 *DECLARE_DYNAMIC_CLASS(class)*宏和*IMPLEMENT_DYNAMIC* CLASS(class, baseClass)宏.另外,如果你希望使用动态创建功能,你还需要保证你的类拥有一个默认的构造函数,否则在编译那些用来动态产生某个对象的代码时, 编译器可能会报错.

下面是定义和动态创建一个对象的例子:

```
class MyRecord: public wxObject
{
DECLARE_DYNAMIC_CLASS(MyRecord)
public:
    MyRecord() {}
    MyRecord(const wxString& name) { m_name = name; }
    void SetName(const wxString& name) { m_name = name; }
    const wxString& GetName() const { return m_name; }
private:
    wxString m_name;
};
IMPLEMENT_DYNAMIC_CLASS(MyRecord, wxObject)
MyRecord* CreateMyRecord(const wxString& name)
{
    MyRecord* rec = wxDynamicCast(wxCreateDynamicObject(wxT("MyRecord")), MyRecord);
    if (rec)
        rec->SetName(name);
    return rec;
}
```

当CreateMyRecord被调用的时候,wxCreateDynamicObject负责创建所需的对象,而wxDynamicCast则负责进行类型检查,如果检查失败则返回NULL.也许你觉得这个代码看上去并没有什么实际的用处,但是在从文件加载一堆不同类型的对象的时候,这是非常有用的.对象的数据和它的名称被一起存放在文件中,通过名字创建一个对象的实例,然后再使用这个实例加载相应的数据.

下面我们来介绍以下运行期类型信息的一起其它相关宏:

CLASSINFO(class)返回一个指向wxClassInfo类型的指针.你可以使用wxObject::IsKindOf函数来判断某个对象是否是对应的类型:

```
if (obj->IsKindOf(CLASSINFO(MyRecord)))
{
    ...
}
```

使用DECLARE_ABSTRACT_CLASS(class)和IMPLEMENT_ABSTRACT_CLASS(class, baseClass)来定义虚类.

使用DECLARE_CLASS2(class)和IMPLEMENT_CLASS2(class, baseClass1, baseClass2)来定义有两个父类的类.

使用DECLARE_APP(class)和IMPLEMENT_APP(class)来使得wxWidgets知道整个应用程序类的运行期类型信息.

wxConstCast(ptr, class)用来代替const_cast<class *>(ptr),如果编译器不支持const_cast,则使用旧的C语言的类型强制转换.

wxDynamicCastThis(class)相当于wxDynamicCast(this, class), 但是后者在某些编译器上会导致永真比较告警,因为它将测试是否指针为空,而this指针永远不可能为空,前者可以避免这个告警.

wxStaticCast(ptr, class)在调试模式将检查强制类型转换的有效性(如果wxDynamicCast(ptr, class) == NULL将引发断言错误)然后返回static_cast<class*>(ptr)的等同结果.

wx_const_cast(T, x)在编译器支持的情况下等同于const_cast<T>(x),否则等同于(T)x(C语言类型强制转换语法).和 wxConstCast不同的是,它强制转换的是T本身而不是指向T的指针,而且参数的顺序也和标准强制转换的顺序相同.

wx_reinterpret_cast(T, x)则相当于reinterpret_cast<T>(x),如果编译器不支持则等同于(T)x.

wx_static_cast(T, x)等同于static_cast<T>(x),如果编译器不支持则等同于(T)x. 和wxStaticCast不同的是,它不做类型检查,并且采用和标准的静态转换相同的参数顺序,而且转换的也是T而不是指向T的指针.

15.6 使用wxModule

wxWidgets的模块管理系统是一个很简单的系统,它允许应用程序(以及wxWidgets自己)可以定义将被wxWidgets自动在开始和退出时执行的初始化和资源清理代码.这有助于避免应用程序在OnInit函数和OnExit函数中依它们功能的需要添加过多的代码.

要定义一个这样的模块,你需要实现一个wxModule的派生类,重载其OnInit和OnExit函数,然后在其声明部分使用 DECLARE_DYNAMIC_CLASS宏,在其实现部分使用 IMPLEMENT_DYNAMIC_CLASS宏(它们可以位于同一个文件内). 在系统初始化的时候,wxWidgets会找到所有wxModule的派生类,创建一个它的实例然后执行其OnInit函数,而在系统退出时执行其 OnExit函数.

举例如下:

```
// 下面这个模块用来自动进行DDE的初始化和清除动作.
class wxDDEModule: public wxModule
{
DECLARE_DYNAMIC_CLASS(wxDDEModule)
public:
    wxDDEModule() {}
    bool OnInit() { wxDDEInitialize(); return true; };
    void OnExit() { wxDDECleanUp(); };
};
IMPLEMENT_DYNAMIC_CLASS(wxDDEModule, wxModule)
```

15.7 加载动态链接库

如果你想要使用位于动态链接库中的函数,你需要使用wxDynamicLibrary类.使用方法是:在其构造函数或者Load函数中传递动态链接库的文件名,如果你不希望wxWidgets自动增加类似.dll(windows)或者.so(linux)这样的扩展名,还需要传递 wxDL_VERBATIM参数.如果加载成功,就可以使用GetSymbol函数通过函数名使用其中的函数了.下面的例子演示了怎样加载和使用 windows的标准控件动态链接库:

```
#include "wx/dynlib.h"
INITCOMMONCONTROLSEX icex;
icex.dwSize = sizeof(icex);
icex.dwICC = ICC_DATE_CLASSES;
// 加载comctl32.dll
wxDynamicLibrary dllComCtl32(wxT("comctl32.dll"), wxDL_VERBATIM);
// 定义ICCEx_t类型
typedef BOOL (WINAPI *ICCEx_t)(INITCOMMONCONTROLSEX *);
// 获取InitCommonControlsEx符号表
ICCEx_t pfnInitCommonControlsEx =
    (ICCEx_t) dllComCtrl32.GetSymbol(wxT("InitCommonControlsEx"));
// 调用获取的函数.
if ( pfnInitCommonControlsEx )
{
    (*pfnInitCommonControlsEx)(&icex);
}
```

wxDYNLIB_FUNCTION宏使得上面代码中的GetSymbol行更为简洁:

```
wxDYNLIB_FUNCTION( ICCEx_t, InitCommonControlsEx, dllComCtl32 );
```

wxDYNLIB_FUNCTION使得你只需要使用一次返回值类型作为其第一个参数,它将创建一个对应的变量,变量名由函数名和pfn前缀组成.

如果动态链接库被成功加载,它将在对应的wxDynamicLibrary对象被释放的时候,在其析构函数中自动被卸载,如果你希望在 wxDynamicLibrary被释放以后继续使用相应的函数,你应该调用首先调用wxDynamicLibrary的Detach函数.

15.8 异常处理

wxWidget创立的时间比起"异常"的概念引入C++要早的多,因此在其代码中已经花费了大量的力气来应付各种各样的异常,因此,可以说,整个 wxWidgets框架内部都没有使用C++的异常机制.当然,这并不意味着你不可以在你的代码中使用C++的异常机制,相反,你在你的代码中使用它是安全的,而且wxWidgets也会帮助你这样作.

要在你的程序中使用异常处理机制,最简单的方法就是根本忽略它的存在,既然wxWidgets不会抛出任何异常,你又何必去处理异常呢?除非你的代码自己抛出了一些异常.这是最简单的方法,但是有时候,对于处理各种可能遇到的错误来说,这种方法是不够的.

另外一个策略是你只用异常机制来处理那些非常致命的系统错误.这种情况下,你不寄希望于你的程序可以从这种致命错误中恢复,它所作的事情只是让你的程序以一种更绅士的方式结束.这种情况下,你只需要重载你的wxApp派生类的OnUnhandledException函数来执行资源清除工作,注意这时候所有和异常有关的信息已经被清除了.如果你需要这些信息,你需要在OnRun函数中针对调用基类函数的语句使用try/catch语句块. 这将使得你可以捕获在应用程序主循环中引发的异常.如果你还希望处理在应用程序初始化和退出时候引发的异常,你需要在你的OnInit和OnExit函数中使用try/catch语句.

最后,如果你希望在异常发生的时候,你的应用程序可以从异常中恢复并且继续运行,那么:如果你程序的异常主要集中在某个类(或其派生类)的事件处理函数中,你可以你可以在这个类的ProcessEvent函数中统一处理这些异常,如果这是不切实际的,你还可以考虑重载wxApp::HandleEvent函数,它将允许你拦截并处理任何由事件处理函数引发的异常.

wxWidgets的异常处理机制默认是打开的,它取决于wxUSE_EXCEPTIONS标记被设置为1.但是如果它被设置为0,在 windows版本中,你需要修改include/wx/msw/setup.h将其更改为1,或者在别的平台上运行configure时增加-- enable-exceptions开关.而将其设置为0或者使用--disable-exceptions开关将会产生更为小巧和相对快速的 wxWidgets库.另外,在windows平台下,如果你使用的是Visual C++,你希望使用wxApp::OnFatalException函数来处理异常而不是引发一个GPF(一般保护性错误),你可以在你的setup.h 中将wxUSE_ON_FATAL_EXCEPTION设置为1.相反的,如果你宁愿将这种错误扔给你的调试器,将它设置成0.

wxWidgets自带一个使用异常的例子,位于samples/except目录中.

15.9 调试提示

自我保护的程序,错误报告何其它的编码技术只能帮助你这么多了,要单步跟踪你的代码,检查每个变量的值,准确的告诉你你的程序有什么异常行为或者从代码的什么地方退出,你还需要使用一个调试工具.因此,针对你的代码,你至少需要维护两套配置文件,一套调试版本何一套正式版本.调试版本将包含更多的错误检测,将关闭编译器的优化开关并且将包含调试程序需要的文件名,行号等调试信息.在调试模式,宏**WXDEBUG**总是被定义了,因此你可以使用这个宏来编写那些仅存在于调试版本中的代码,不过类似wxLogDebug这样的函数,即使你没有使用**WXDEBUG**宏将其包含,它仍然将在正式版本中被移除.

确实有很多人从来没有使用过调试器,但是在你熟悉了这些工具以后,它将大大降低你的工作量.在windows平台上,VC自带了一个很不错的调试器.如果你使用的是GCC,你可以使用GDB工具包,它工作在命令行模式下,你也可以使用一些集成了GDB的IDE环境.更多的信息可以参考附录E,"wxWidgets的第三方工具".

wxWidgets支持同时编译多个版本.在windows平台上,你可以给对应的Makefile传递BUILD=debug或 BUILD=release这样的开关.如果你使用的是configure程序,你可以配置不同的版本编译在不同的目录,然后在各个版本中使用类似-- enable-debug或--disable-debug这样不同的开关.某些集成开发环境出于各种各样的原因不允许你的应用程序同时使用不同的配置文件,对于这样的集成开发环境,作者的忠告是:不要使用它们.

调试X11错误

极少的情形下,wxGTK程序会由于X11的错误而崩溃,这时候你的应用程序将立即退出而不给你任何栈调用情况的打印,这种问题是非常难以跟踪的,在这种情况下,你需要象下面代码展示的那样,增加一个错误处理函数:

```
#if defined(__WXGTK__)
#include &lt;X11/Xlib.h&gt;
typedef int (*XErrorHandlerFunc)(Display *, XErrorEvent *);
XErrorHandlerFunc gs_pfnXErrorHandler = 0;
int wxXErrorHandler(Display *display, XErrorEvent *error)
{
    if (error->error_code)
    {
          char buf[64];
          XGetErrorText (display, error->error_code, buf, 63);
          printf ("** X11 error in wxWidgets for GTK+: %s\n  serial %ld error_code %d
 request_code %d minor_code %d\n",
           buf,
           error->serial,
           error->error_code,
           error->request_code,
                   error->minor_code);
    }
    // 去掉下面的注释以便将错误重定向道你的处理函数.
#if 0
    if (gs_pfnXErrorHandler)
        return gs_pfnXErrorHandler(display, error);
#endif
    return 0;
}
#endif
  // __WXGTK__
bool MyApp::OnInit(void)
{
#if defined(__WXGTK__)
    // 安装错误处理函数
    gs_pfnXErrorHandler = XSetErrorHandler( wxXErrorHandler );
#endif
    ...
    return true;
}
```

现在,你的应用程序在遇到这样的错误的时候,将会产生一个普通的段错误.如果你在启动你的应用程序的时候传递了--sync参数,这个段错误将正好显示在被传递了错误参数的X11函数的地方.

一个简单有效的定位问题方法

如果你确实碰到了一个很难定位的问题,一个好的方法是使用尽量少的代码来重现这个问题,你可以修改wxWidgets自带的任何一个例子,增加一些代码来重现你的问题,或者把你的代码制作一份拷贝,逐段的去掉那些不影响这个错误的代码,直至你可以准确的定位出是那些代码导致了这个错误的产生.如果你认为这是wxWidgets本身的错误,把你修改后的导致问题出现的wxWidgets例子发送给wxWidgets社区,相信这个问题将被很快修正.

调试一个发布版本

某些情况下,你的应用程序可能在调试版本工作正常,而在正式版本中则工作不正常,这可能是由于编译器使用的不同运行期库文件的细微差别导致的.如果你正在使用的是VC,你可以创建一个和调试版本一模一样的配置,但是却定义了NDEBUG宏,这将使得你的代码和wxWidgets和调试信息都具备,运行的时候却使用的是发布版本的运行库.这至少可以让你确定是不是由于运行期库的原因导致了这个问题.

通常当你发布你的应用程序的时候都将使用去除了所有调试信息的版本,但是有时候你的客户会遇到一些在你的机器上很难重现的问题,,这时候你可能想给你的用户发送一份调试版本的程序(在windows平台上,通常你需要使用静态编译的方法以避免同时发送那些调试模式的动态链接库).然后在你的客户的机器上,可以使用一个叫做Dr.Watson的程序来运行你的包含调试信息的程序,在程序异常退出的时候,将会产生一个文件记录当时的情况.参考你的编译器的信息以及网上的教程来了解怎样使用由Dr.Watson产生的这个文件来定位你的代码中的异常.

如果你使用的是MinGW,你可以使用一个叫做Dr.MinGW的工具来在程序异常退出时候打印出一个有用的当前函数调用堆栈(当然, 如果你的代码包含调试信息(打开了-g开关)的话).你可以从 http://www.mingw.org 下载这个工具,如果你的客户有耐心并且很合作,你可以把这个工具发送给他们然后让他们把产生的报告发送给你.

在Unix平台上,调试版本的可执行文件可以产生一个core文件(这依赖于系统的设置,参考Unix命令ulimit的man手册).你可以象你平时调试可执行文件那样使用这个core文件,以便观察程序退出时候的现场情况.然后,这个core文件可能会很大,你的客户可能不大愿意发送给你这样的一个core dump文件.

另外的一个替代方案是,你可以在你的程序中自己记录程序的重要的执行情况.这方面你可以参考wxWidgets手册中 wxDebugReport类相关的内容,这个类可以产生一个合适email给程序开发商的报告.类似的,你还可以使用来自http: //wxcode.sf.net的wxCrashPrint(for linux)或者来自 http://www.codeproject.com/tools/blackbox.asp 的BlackBox(for windows).

第十五章小结

在这一章里,我们讨论了内存管理和错误检测各个方面的内容.你现在应该了解到,什么时候使用new来创建对象,什么时候直接使用栈变量,你的应用程序应该怎样进行资源清除工作,怎么识别内存泄漏,怎么使用宏来创建有自我保护意识的程序.你也看到了怎样动态创建对象,也知道了何时使用wxLogDebug何时使用wxLogError.也看到了怎样在wxWidgets中使用C++异常机制,我们还对怎样调试应用程序给出了一些提示. 接下来我们来看看怎样让你的应用程序支持多种语言.

第十六章编写国际化程序

如果你象要求你的程序支持多平台一样要求你的程序支持多语言,那么你的程序可能会有更多的用户,这将很大程度的提高你的程序成功的机会.本章我们就来谈一谈怎样实现这样的目标,要让你的应用程序成为国际化的程序,你需要用到的知识,也就是俗称的"i18n"(没错,i18n).

16.1 国际化介绍

当把你的程序带到国际化舞态上时,想到的第一件事情就是翻译.你需要为你的应用程序中的每一个字符串针对每一种可能用到的语言准备一个翻译字符串.在 wxWidgets中,这是通过使用wxLocale类来加载某个分类条目来实现的.这种技术也许和你以前用过的字符串表的方式不太一样,字符串表的方式是给程序中的每个字符串一个标识符,然后通过切换不同的字符串表来实现翻译的.而分类条目则是通过把字符串按照某一种分类进行翻译的作法(你当然可以使用你的本地系统支持的任何方法,不过要注意,在wxWidgets库内部,也使用的是分类条目的方法).

如果没有指定分类条目,源代码中的字符串将被显示在用户界面的菜单或者按钮等控件上.如果你的本地语言包含非ASCII字符,你就必须进行翻译(使用一个分类条目),因为源代码只支持ASCII码.

代表另外一种语言的文本可以采用各种各样的编码方式,这意味着同一个字节在不同的语言中可能代表不同的字符.你需要确定你的应用程序能够正确识别这些编码并且你的GUI控件可以正确显示这些编码.也就是说,你需要注意各种情况下使用的编码方式以及不同编码方式之间的互相转换的问题.

国际化的另外一个方面是格式化数字,日期和时间.注意即使是在同一种语言中,它们也有可能不同.比如英语中的数字1,234.56在德语中被写作1.234,56,而在瑞士德语中则被写作1'234.56.在美国11月10日的表示方法是11/10,而同样的写法在UK则代表10月11 日.我们很快会介绍wxWidgets如何处理这些复杂的情况.

翻译的字符串有时候会比代码中的英文字符串要长,这意味着窗口控件的布局需要调整为不同的大小.在第7章(使用布局控件进行布局)中介绍的布局控件可以解决这个问题.另外一个关于布局的问题是,英语是从左向右阅读的,而某些语言,比如阿拉伯语和希伯来语是从右到左的顺序读的(称为 RTL),目前wxWidgets中对于这个问题还没有一个很好的解决机制.

最后一个需要根据不同的语言进行调整的项目是你用到的图片和声音.比如说,你正在制作一个电话目录程序,它有一个特性是可以读出来电话号码,这个读音是语言有关的,另外你也可能需要根据国家的不同使用不同的图片.

16.2 从翻译说起

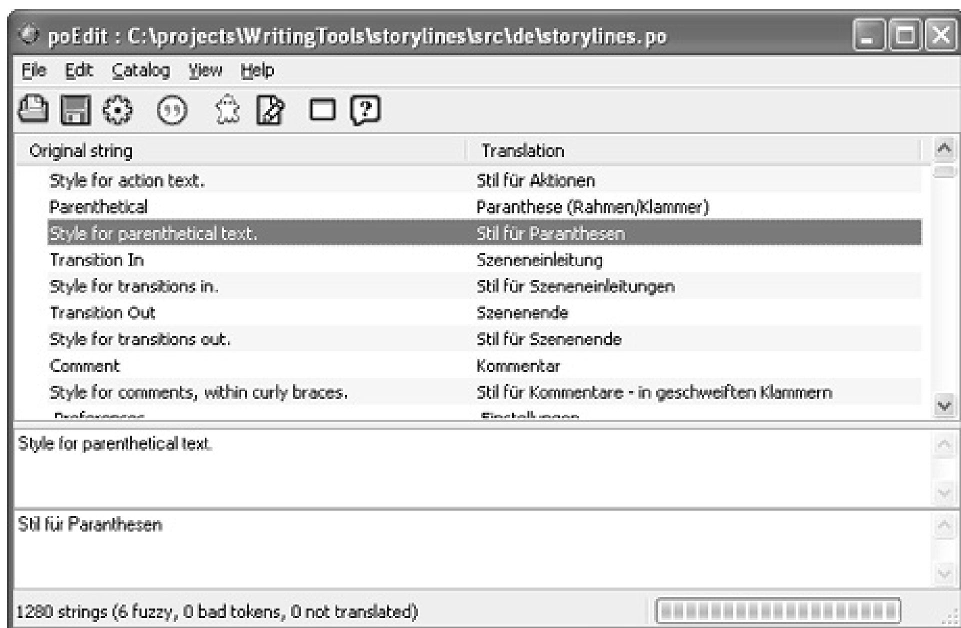
wxWidgets是通过wxLocale来提供语言翻译支持的.而且它自己也已经被完整的支持了很多种语言.请从wxWidgets的官方网站下载最新的翻译包.

wxWidgets提供的国际化语言支持和GNU的gettext工具包非常的相似.wxWidgets使用的分类条目机制和gettext的分类条目机制是二进制兼容的,意味着你可以使用所有的gettext工具.并且在运行时不需要别的任何附加的库支持因为wxWidgets内部实现了对条目文件的读取.

在程序开发过程中,你需要gettext工具包来操作分类条目(或者poEdit工具,参考下面的小节).有两种类型的消息分类条目文件,一种是源文件,它是文本格式文件,扩展名为.po,另外一种是二进制分类条目文件,它是通过分类条目源文件使用gettext工具包中的工具 msgfmt创建的,扩展名是.mo. 在程序运行过程中只需要二进制的分类条目文件.针对每一种你想要支持的语言,你都需要单独提供种分类条目文件.

poEdit

你不需要使用gettext提供的命令行工具来维护你的分类条目.Vaclav Slavik制作了一个叫做poEdit的工具,它是一个图形化的gettext前端工具,可以从 http://www.poedit.org 下载.运行界面如下图所示.它可以用来帮助你维护分类条目,产生.mo文件以及随着你的代码的更改和增加将新的需要翻译的条目合入旧的分类条目文件.



一步一步介绍创建消息翻译分类条目

遵循下面的这些步骤来进行某个消息条目的建立:

1. 在你的代码中,将需要翻译的字符串常量使用wxGettranslation宏或者它的简短替代品_()宏括起来.对于那些不需要翻译的字符串,也请使用wxT()或者等价的_T()宏括起来,以便其

可以实现Unicode兼容.

2. 将你在代码中标识为需要翻译的字符串整理到.po文件中.当然,这么负责的步骤你不需要使用手工去作它,你可以使用gettext工具包中的 xgettext工具,然后使用"-k"参数指定到底翻译wxGettranslation宏还是_()宏指代的字符串. poEdit工具也可以帮你完成这个工作,和xgettext的"-k"参数等价的功能需要在配置对话框中指定.
3. 将整理好的.po文件中的字符串翻译为你希望使用的那种语言的字符串,每种语言对应一个.po文件,并且要指定这种语言使用的编码方式.

   如果没有使用poEdit工具,你可以使用任何你喜欢的文本编辑器来完成这个工作,对应的.po文件的文件头如下所示:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR &lt;EMAIL@ADDRESS&gt;, YEAR.
#
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 1999-02-19 16:03+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME &lt;EMAIL@ADDRESS&gt;\n"
"Language-Team: LANGUAGE &lt;LL@li.org&gt;\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=iso8859-1\n"
"Content-Transfer-Encoding: 8bit\n"
```

   注意倒数第二行的charset属性,指定了这个分类条目文件使用的编码方式.本文件中所有的字符串都将采用这种编码方式,这是非常重要的,因为如果你使用了非Unicode的编码而没有指定其编码方式,那些GUI控件将不知道怎样显示你翻译的文本.

4. 将翻译好的.po文件编译为二进制的.mo文件.这要用到一个工具叫做msgfmt,poEdit也可以帮你完成这个工作. 使用msgfmt的命令行如下所示:

```
msgfmt -o myapp.mo myapp.po
```

5. 在你的代码中设置合适的locale参数以便它使用对应的分类条目(我们将在接下来的小节,"使用wxLocale"中介绍相关内容).

在Mac OS X系统上,你还需要更改一个叫做Info.plist的文件,这个文件是用来描述你的软件包的相关信息的.它是一个使用UTF-8编码的Xml文件,其中包含一个叫做CFBundleDevelopmentRegion的条目用来描述软件的开发语言(比如说:英语),而Mac OSX将会查询软件包中某些默认路径来找到这个软件支持的语言.例如,如果存在目录German.lproj,则认为这个软件支持德语.因为 wxWidgets并不使用这样的目录名称,你需要在这个文件中显式的指定你的应用程序支持的语言类型.这可以通过在其中增加 CFBundleLocalizations条目来实现,如下所示:

```
<key>CFBundleDevelopmentRegion</key>
<string>English</string>
<key>CFBundleLocalizations</key>
<array>
       <string>en</string>
       <string>de</string>
       <string>fr</string>
</array>
```

使用wxLocale

wxLocale封装了所有和本地化相关的设置,类似于C语言中的locale的概念.通常你在你的应用程序类中定义一个wxLocale类型的成员,比如说:m_locale,然后在你的应用程序的OnInit函数中,象下面这样初始化这个变量:

```
if (m_locale.Init(wxLANGUAGE_DEFAULT,
                    wxLOCALE_LOAD_DEFAULT | wxLOCALE_CONV_ENCODING))
{
    m_locale.AddCatalog(wxT("myapp"));
}
```

注意wxLocale::Init函数将会查找wxstd.mo文件,这个文件代表wxWidgets自己的翻译分类条目.参数 wxLANGUAGE_DEFAULT指定使用系统默认的语言,你也可以通过使用对应的wxLANGUAGE_xxx宏来强制指定某种特定的语言.

当wxLocale加载一个翻译条目的时候,这个条目被自动从它自己的编码转换成系统当前使用的编码,这是wxLocale的默认行为,如果你不希望使用这个行为,你可以在调用wxLocale::Init时不传递wxLOCALE_CONV_ENCODING标记.

wxWidgets在哪些目录里寻找对应的.mo文件呢?对于任何一个待查目录<DIR>,它的查找范围包括下面的这些目录:

```
<DIR>/<LANG>/LC_MESSAGES
<DIR>/<LANG>
<DIR>
```

到底哪些目录是待查目录,依系统的不同各不相同:

- 在所有的平台上,LC_PATH环境变量指定的目录将成为待查目录.
- 在Unix或Mac OS X上,wxWidgets的安装路径将成为待查目录,另外还有/share/locale, /usr/share/locale, /usr/lib/locale, /usr/locale /share/locale以及当前目录.
- 在Windows平台上,应用程序所在的目录也将成为待查目录.

你还可以通过函数wxLocale::AddCatalogLookupPathPrefix增加你自己的待查目录,比如:

```
wxString resDir = GetAppDir() + wxFILE_SEP_PATH + wxT("resources");
m_locale.AddCatalogLookupPathPrefix(resDir);
// 假设resDir是c:\MyApp\resources, 假设当前使用法语,
// AddCatalog将查找的待查目录除了前面介绍的所有待查目录外,
// 还将额外查找下面的路径:
//
// c:\MyApp\resources\fr\LC_MESSAGES\myapp.mo
// c:\MyApp\resources\fr\myapp.mo
// c:\MyApp\resources\myapp.mo
m_locale.AddCatalog(wxT("myapp"));
```

通常在发行你的应用程序的时候,你应该为每个语言创建一个子目录,目录名称使用代码某种区域的标准的国际化名称,然后在其中放置对应的 <appname>.mo.比如,wxWidgets自带的internat例子程序将ISO639格式编码的法语和德语翻译文件分别放置在 fr和de目录中.

16.3 字符编码和Unicode

这个世界上有太多太多的字符,远超过了一个字节(8bit)可能容纳的256个数目.为了显式超过256个字符以外的其它字符,一个新的手段被增加进来,那就是字符编码和字符集(更新和更好的"Unicode"解决方案,我们也将很快谈到.).

因此,到底字节161代表什么字符,是由当前使用的字符集决定的.在ISO 8859-1(Latin-1)字符集中,它代表的是一个倒写的感叹号,而在ISO 8859-2字符集中,则代表的是字母a(Aogonek).

当你在一个窗口上绘制字符的时候,系统必须知道你使用的编码,这成为字体编码,也就是所谓的字符集.创建一个没有指定字符集的字体意味着使用默认编码,这在大多数系统上都是没有问题的,因为大多数人都在使用支持本国语言的系统.

但是,如果你确定某些字符使用的是不同的编码(比如ISO 8859-2),在创建字体的时候,你应该指定这种编码,如下所示:

```
wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
              false, wxT("Arial"), wxFONTENCODING_ISO8859_2);
```

否则,在一个西文系统ISO 8859-1中,字符将不能被正确显式.

有时候可能我们无法找到一个合适的满足某种编码的字体,这种情况下,我们可以尝试使用一种代替字体,不过你需要将要显式的字体转换成那种代替字体对应的编码方式.下面的代码演示了应该怎样作.一个字符串text的编码为enc,准备用字体facename显示.同时下面的代码也演示了wxCSConv的用法:

```
// 我们有一段'enc'编码的文本,我们希望用字体
// 'facename'显示.
//
// 首先,我们必须确定这个字体可以显示这种编码
wxString text; // 编码方式为 'enc'
if (!wxFontMapper::Get()->IsEncodingAvailable(enc, facename))
{
   // 不能支持这种编码,需要查找替代编码.
   // 能支持某种替代编码吗?
   wxFontEncoding alternative;
   if (wxFontMapper::Get()->GetAltForEncoding(enc, &alternative,
                                              facename, false))
   {
       // 我们找到了替代编码方案'alternative',
       // 因此我们进行编码的转换,转换成alternative.
       wxCSConv convFrom(wxFontMapper::GetEncodingName(enc));
       wxCSConv convTo(wxFontMapper::GetEncodingName(alternative));
       text = wxString(text.wc_str(convFrom), convTo) ;
       // 然后创建alternative编码的字体
       wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
               false, facename , alternative);
       dc.SetFont(myFont);
   }
   else
   {
      // 不能找到完美替代编码;尝试有损耗的编码方案
      // ISO 8859-1 (7-bit ASCII)
      wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
              false, facename, wxFONTENCODING_ISO8859_1);
      dc.SetFont(myFont);
    }
}
else
{
    // OK,这个字体可以支持这个编码.
     wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
             false, facename, enc);
     dc.SetFont(myFont);
}
// 最后,我们使用选择的字体绘制可能已经经过编码转换的字符串.
dc.DrawText(text, 100, 100);
```

转换数据

前面的代码演示了将一组字节流从一种编码转换为另外一种编码的方法.这种转换可以有两种方法,第一种是使用 wxEncodingConverter类,这种方法是不被推荐的(可能在后续版本种被淘汰的方法),你不应该在新的代码种使用这种方法,除非你的编译器不支持wchar_t结构. 推荐使用第二种方法,字符集转换(使用基于wxMBConv的wxCSConv).

wxEncodingConverter

这种方法只能支持部分的字符集,但是如果你的编译器不支持wchar_t结构,这是你唯一的选择,转换方法如下:

```
wxEncodingConverter converter(enc, alternative, wxCONVERT_SUBSTITUTE);
text = converter.Convert(text);
```

wxCONVERT_SUBSTITUTE标记表明允许转换过程中如果找不到严格对应的字符,允许存在信息损失, 这将导致带重音符号的字母变成普通的字母或者短破折号和长破折号统一用"-"来代替等.

wxCSConv (wxMBConv)

Unicode的解决方案的核心是,它使用16bit或者甚至是32bit的wchar_t结构来代表一个字符,因此它可以把全世界所有的字符用一种编码表示.这意味着你不需要处理任何编码转换之类的问题除非你需要处理老的8-bit格式数据,前面我们已经说过,8bit的数据必须和字符集一起使用才有意义.

即使你没有把wxWidgets编译成Unicode模式(这种模式下,所有的字符串都是Unicode编码格式),只要你的系统支持,你还是可以使用它进行编码转换.转换的方法是,先把你的字符串从它的编码转换成Unicode编码,然后再从Unicode编码转换成目标编码. wxString类也使用这种方法来提供编码转换支持.要记住的是:非Unicode版本的wxWidgets中的wxString对象采用的是8bit的方法保存字符串,因此它自己并不知道其内部的数据使用的是什么编码方式.

如果想把wxString转换成Unicode,你需要使用wxString::wc_str函数,这个函数采用一个多字节转换类作为它的参数,这个参数告诉非Unicode版本的wxString它内部的字符串是采用什么编码方式的,但是在Unicode版本的wxWidgets中, 这个参数被忽略,因为wxString内部的编码已经是Unicode了.

在Unicode版本中,我们可以直接使用wx_str返回的字符串了,但是在非Unicode版本中,我们还需要将其转换为我们可以支持的编码方式convTo,因此在下面的代码中,在Unicode版本中,convTo也将被忽略:

```
text = wxString(text.wc_str(convFrom), convTo);
```

可以看到字符集编码比字体字体编码更常使用,因此有时候你需要通过下面的代码将字体编码名字装换成字符集编码名字:

```
wxFontMapper::GetEncodingName(fontencoding);
```

这就是上面例子中下面这一部分代码的含义:

```
wxCSConv convFrom(wxFontMapper::GetEncodingName(enc));
wxCSConv convTo(wxFontMapper::GetEncodingName(alternative));
text = wxString(text.wc_str(convFrom) , convTo) ;
```

有时候你需要直接使用8bit的字节流而不是使用wxString,这可以通过使用wxCharBuffer类获得,下面我们看看这一行代码:

```
wxCharBuffer output = convTo.cWC2MB(text.wc_str(convFrom));
```

如果你的输入数据不是一个字符串而也是一个8bit的数据流(比如也是一个wxCharBuffer),你可以使用下面的转换方式:

```
wxCharBuffer output = convTo.cWC2MB(convFrom.cMB2WC(input));
```

wxWidgets定义了一些全局的类用于实现字符转换,比如wxConvISO8859_1是一个对象,而wxConvCurrent 是一个指针,指向当前标准C的locale指定的编码类.另外还有一些wxMBConv的子类用来优化特定的编码转换任务,比如 wxMBConvUTF7,wxMBConvUTF8, wxMBConvUTF16LE/BE和wxMBConvUTF32LE/BE.其中后两个被重定义为wxMBConvUFT16/32,它使用机器本身的字节序.更多信息请参考wxWidgets手册中的"wxMBConv Classes Overview"小节.

转化来自外部的临时缓存数据

正如我们刚刚讨论的那样,转换类允许你很方便的把一种字符集转换为另外一种字符集.然而,大多数的转换结果为一个新创建的字符串或者一个临时缓存.有时候我们需要将转换的结果保存起来已备以后使用,这种情况下我们可以把转换的结果复制到一个独立的存储区.

假设我们想在两个电脑之间通过socket传递字符串.我们首先应该在字符串采用的编码上取得一致.否则,平台默认的编码可能把传递的字符串搞的一团糟.在我们的这个例子中,我们把发送出去的字符串先转换成UTF-8编码,在接收的部分,在将UTF-8编码的字符串转换成系统默认的字符串.

下面的代码演示了怎样将符合本地编码的字符串转换成UTF-8,将转换结果存储在一个char*指针中,然后通过socket发送出去,接收的电脑再将收到的字符串从UTF-8转换成它自己的电脑上的本地编码.

```
// 将本地编码字符串转换成UTF-8编码
const wxCharBuffer ConvertToUTF8(wxString anyString)
{
    return wxConvUTF8.cWC2MB( anyString.wc_str(*wxConvCurrent) ) ;
}
// 将UTF-8编码的字符串转换成本地编码字符串
wxString ConvertFromUTF8(const char* rawUTF8)
{
    return wxString(wxConvUTF8.cMB2WC(rawUTF8), *wxConvCurrent);
}
// 测试以下这两个转换函数
void StringConversionTest(wxString anyString)
{
    // 转化成UTF-8编码并保存在wxCharBuffer中.
    const wxCharBuffer bUTF8 = ConvertToUTF8(anyString);
    // wxCharBuffer可以隐式的转换成char*.
    const char *cUTF8 = bUTF8 ;
    // 重建字符串
    wxString stringCopy = ConvertFromUTF8(cUTF8);
    // 因为是同一个电脑,这两个字符串应该是完全相同的.
    wxASSERT(anyString == stringCopy);
}
```

帮助文件

你需要为每个支持的语言制作一份帮助文件.你的帮助文件控制器在初始化的时候将指定帮助文件的名称.你可以使用wxLocale::GetName来获取语言相关的名称,也可以直接使用前面介绍的_()宏以便获得语言相关的名称.比如:

```
m_helpController->Initialize(_("help_english"));
```

如果你使用的是wxHtmlHelpController,记住你需要给每一个帮助页面指定META标记,如下所示:

```
<meta http-equiv="Content-Type" content="text/html; charset=iso8859 //2">
```

你还需要注意帮助工程文件(扩展名.hhp)也许要包含一个指定编码的选项行:

```
Charset=iso8859-2
```

这个额外的条目告诉HTML帮助控制器帮助内容和帮助索引使用什么编码格式编码的.

16.4 数字和日期

本地化程序中的另外一个方面是格式化数字和日期,对于数字,基于printf的wxString格式化函数已经在内部实现了针对不同地域的本地化,如下面的代码所示:

```
wxString::Format(wxT("%.1f") , myDouble);
```

这里,Format函数将会根据你设置的locale帮你处理地域差异. 而下面的日期格式化代码:

```
wxDateTime t = wxDateTime::Now();
wxString str = t.Format();
```

Format函数也将根据你设置的locale进行合适的格式化操作.在wxWidgets手册中时间和日期函数格式化的相关部分详细的介绍了怎样根据自定义的格式进行时间和日期的格式化.在这种情况下,你只需要将格式化文本使用_()宏包括起来,然后针对不同的语言翻译成对应的本地格式就可以了.

如果你想知道当前设置的locale对应的数字分割符或者别的一些本地化相关的值,可以使用wxLocale的GetInfo函数,比如下面的代码返回当前设置的locale下数字的10进制分割符:

```
wxString info = m_locale.GetInfo(wxLOCALE_THOUSANDS_SEP,
                                   wxLOCALE_CAT_NUMBER) ;
```

16.5 其它媒介

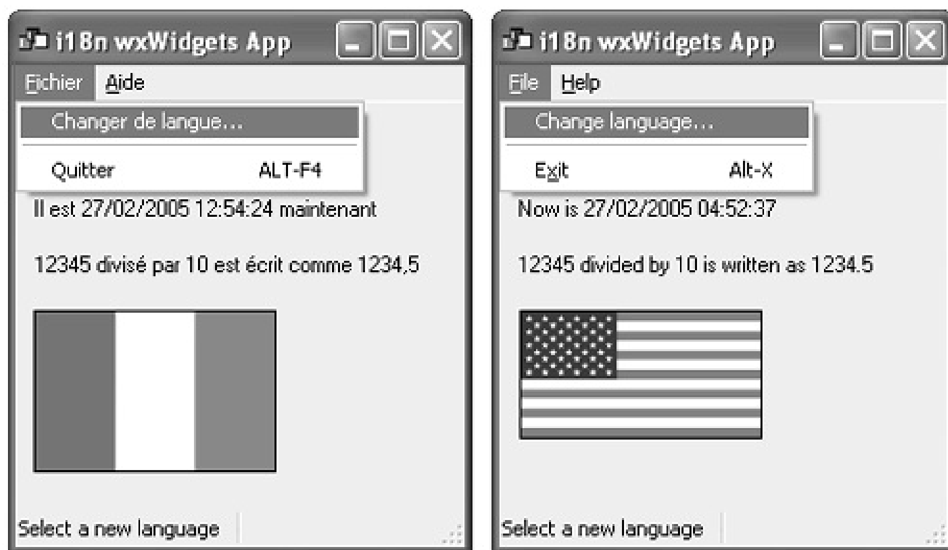
你可以使用和文本同样的机制来针对不同的语言加载不同的声音和图片,如下所示:

```
wxBitmap bitmap(_("flag.png"));
```

上面的代码将导致flag.png在运行期被翻译,因此你可以将其针对不同的语言翻译成不同的文件,比如de/flag.png.当然,你需要保证这是一个存在的真实文件.你也可以使用第14章,"文件和流"中介绍的技术将其翻译为一个虚拟文件系统路径.

16.6 一个小例子

为了演示本章介绍的这些内容,随书光盘上examples/chap16目录中举了一个小例子.它以三种语言显示了一些字符串和图片:英语,法语和德语. 你可以从文件菜单更改当前的语言,这将导致菜单字符串,静态文本控件和使用的图片作出相应的改变.为了演示_()宏和wxT()的区别,状态栏的字符串始 终保持英语不变.



这个例子的应用程序类包含一个指向wxLocale类型的指针和一个函数SelectLanguage用来更改当前的语言.主要的声明和实现如下:

```
class MyApp : public wxApp
{
public:
    ~MyApp() ;
    // 初始化应用程序
    virtual bool OnInit();
    // 根据用户选择的语言重新创建wxLocale变量
    void SelectLanguage(int lang);
private:
    wxLocale* m_locale; // 'our' locale
};
IMPLEMENT_APP(MyApp)
bool MyApp::OnInit()
{
    wxImage::AddHandler( new wxPNGHandler );
    m_locale = NULL;
    SelectLanguage( wxLANGUAGE_DEFAULT );
    MyFrame *frame = new MyFrame(_("i18n wxWidgets App"));
    frame->Show(true);
    return true;
}
void MyApp::SelectLanguage(int lang)
{
    delete m_locale;
    m_locale = new wxLocale( lang );
    m_locale->AddCatalog( wxT("i18n") );
}
MyApp::~MyApp()
{
    delete m_locale;
}
```

主窗口的两个函数SetupStrings和OnChangeLanguage可能是你最感兴趣的部分,SetupStrings更改相关控件的字符串并且重新创建菜单条,以便演示更改wxLocale以后相关字符串的翻译:

```
void MyFrame::SetupStrings()
{
    m_helloString->SetLabel(_("Welcome to International Sample"));
    m_todayString->SetLabel( wxString::Format(_("Now is %s") , wxDateTime::Now().Format()
.c_str() ) );
    m_thousandString->SetLabel( wxString::Format(_("12345 divided by 10 is written as %
.1f") , 1234.5 ) );
    m_flag->SetBitmap(wxBitmap( _("flag.png") , wxBITMAP_TYPE_PNG ));
    // 创建菜单条
    wxMenu *menuFile = new wxMenu;
    // About菜单应该位于帮助菜单
    wxMenu *helpMenu = new wxMenu;
    helpMenu->Append(wxID_ABOUT, _("&About...\tF1"),
                     wxT("Show about dialog"));
    menuFile->Append(wxID_NEW, _("Change language..."),
                     wxT("Select a new language"));
    menuFile->AppendSeparator();
    menuFile->Append(wxID_EXIT, _("E&xit\tAlt-X"),
                     wxT("Quit this program"));
    wxMenuBar *menuBar = new wxMenuBar();
    menuBar->Append(menuFile, _("&File"));
    menuBar->Append(helpMenu, _("&Help"));
    wxMenuBar* formerMenuBar = GetMenuBar();
    SetMenuBar(menuBar);
    delete formerMenuBar;
    SetStatusText(_("Welcome to wxWidgets!"));
}
```

OnChangeLanguage在用户更改当前语言的时候被调用,它将用户的选择映射到某种语言标识(比如 wxLANGUAGE_GERMAN)上.这个标识被传递给MyApp::SelectLanguage以便设置当前的locale,然后调用 SetupStrings根据设置的locale更改当前的字符串和图片,如下所示:

```
void MyFrame::OnChangeLanguage(wxCommandEvent& event)
{
    wxArrayInt languageCodes;
    wxArrayString languageNames;
    languageCodes.Add(wxLANGUAGE_GERMAN);
    languageNames.Add(_("German"));
    languageCodes.Add(wxLANGUAGE_FRENCH);
    languageNames.Add(_("French"));
    languageCodes.Add(wxLANGUAGE_ENGLISH);
    languageNames.Add(_("English"));
    int lang = wxGetSingleChoiceIndex( _("Select language:"),
                         _("Language"), languageNames );
    if ( lang != -1 )
    {
        wxGetApp().SelectLanguage(languageCodes[lang]);
        SetupStrings();
    }
}
```

第十六章小结

本章我们介绍了怎样处理字符串,时间,日期,货币等在不同语言之间的翻译.最后的忠告是:你应该和一个熟悉那个国家语言的人一起作这个事情,才能了解到不同语言之间一些细微的差别.

wxWidgets自带的samples/internat目录中也有一个相关的例子,它将使用到的字符串翻译成了十种语言.

下一章里,我们将学习怎样让你的应用程序通过多线程的方法同时处理多个任务.

第十七章编写多线程程序

大多数时候,事件驱动的GUI程序可以给你造成一个很好的假象:多个任务在同时的运行.这是因为,重绘窗口通常只占用很少的时间,用户输入也被很快的进行了处理.然后,有时候,有些任务很难将其分割成足够小的让人难以察觉的小块来运行,这时候就要使用多线程编程了.本章我们来介绍一下在wxWidgets 中怎样实现多线程编程.在本章的最后,我们将介绍一下多线程编程的一些替代解决方案.

17.1 什么时候使用多线程,什么时候不要使用

线程基本上来说是你的应用程序中一条单独的执行路径.有些地方把线程成为轻量级的进程,但是线程和进程有着一个最本质的区别,那就是:进程是在不同的地址空间运行的,而同一个进程内的所有的线程都在同一个地址空间运行. 当然这样的好处是,同一个进程的线程互相访问公共的数据是非常方便的,不过这也造成了另外一个经常犯错误的地方,就是很一个数据很容易被多个线程同时访问,造成不可知的后果,因此强烈推荐对于这些数据的访问,必须小心的使用用于同步变量访问的变量,比如信号量和关键区域.

如果使用的当,多线程编程可以简化应用程序的体系结构,并且将用户界面和其后面的真实世界分割开来.注意这通常不会让你的应用程序运行的更快,除非你有多个处理器,但是用户界面通常会变得更灵敏.

wxWidgets既提供了线程的支持,也提供了信号量和带条件的关键区域的支持.其线程API主要参考的是pthread(POSIX线程)模型,不过某些API采用了不同的名字,而另外一些API则吸收了部分Win32线程API的灵感.

这些类使得编写多线程的程序变得简单,并且还提供了相对于本地线程API更多的错误检查.尽管如此,多线程编程仍然不是一个很简单的事情,对于大型的项目尤其如此.因此,在考虑编写新的多线程程序或者在旧的代码中加入多线程之前,仔细的考虑一下是不是应该采取可选的替代方案来实现同样的功能,是一件很值得一做的事情.在有些情况下,线程是唯一最优的选择,比如对于一个FTP服务器来说,为每一个新的连接创建一个线程,但是,如果只是增加一个线程来为某个长时间的运算显示一个进度条来说,这种线程的时候就有点过犹不及了.在这种情况下,你可以将计算任何放在系统空闲的时候进行,并且周期性的调用wxWindow::Update来更新用户界面就可以了.关于这个问题更详细的描述,请参考本章的最后一节,"多线程的替代方案"中的描述.

如果你还是决定在你的代码中使用多线程,我们强烈建议你只在你的主线程中调用所有和GUI有关的函数.尽管wxWidgets自带的线程例子中演示了怎样在多个线程中同时调用GUI函数,但是,一般来说,这是非常非常差劲的设计.一个主线程负责GUI,其它多个线程负责别的计算工作,他们之间通过事件互相通讯,这样的设计是更好的设计并且通常可以避免很多的错误从而节约你大量的调试的时间.比如说,在Win32平台上,线程只能使用由自己而不是别的线程创建的GUI对象(画笔,画刷之类).

要实现线程之间通讯,你可以使用wxEvtHandler::AddPendingEvent函数或者它的简化版本wxPostEvent,这些函数被设计成线程安全的因此你可以使用他们在线程之间发送事件.

17.2 使用wxThread

如果你要在你的代码中使用线程,首先要实现一个wxThread的派生类,并且至少要重载其虚函数Entry,这个函数包含了线程要做的主要的事情.举例来说,比如我们要用一个单独的线程来计算图片中颜色的数目,下面是我们的派生类的声明:

```
class MyThread : public wxThread
{
public:
    MyThread(wxImage* image, int* count):
        m_image(image), m_count(count) {}
    virtual void *Entry();
private:
    wxImage* m_image;
    int*     m_count;
};
// 一个标识符 用来在线程工作完成的时候通知应用程序.
#define ID_COUNTED_COLORS    100
```

Entry函数用来进行计算工作并且返回一个返回值(对于联合线程(即将介绍),Wait函数将返回这个值),下面是我们的Entry函数:

```
void *MyThread::Entry()
{
    (* m_count) = m_image->CountColours();
    // 使用一个已知的事件来通知应用程序.
    wxCommandEvent event(wxEVT_COMMAND_MENU_SELECTED,
                         ID_COUNTED_COLORS);
    wxGetApp().AddPendingEvent(event);
    return NULL;
}
```

为了简单起见,我们没有定义新的事件而是使用了一个已有的事件通知应用程序线程的工作已经做完了.

线程的创建

线程的创建分为两步,首先产生一个线程的实例,然后调用线程的Create函数:

```
MyThread *thread = new MyThread();
if ( thread->Create() != wxTHREAD_NO_ERROR )
{
    wxLogError(wxT("Can't create thread!"));
}
```

有两种不同的线程,一种线程你在启动之后就可以忘记它的存在,而另外一种,你需要等待它返回一个结果.前者我们称为分离线程,后者称为联合线程. 线程的类型是通过调用wxThread的构造函数时传递的参数是wxTHREAD_DETACHED还是wxThrEAD_JOINABLE来决定的,联合线程的返回值是由对其成员函数Wait的调用返回的,而对于分离线程,不可以调用Wait函数.

你不必把所有的线程都创建为联合线程,,因为联合线程有它不方便的地方,你必须调用联合线程的Wait函数来等待联合线程结束,否则系统为这个线程分配的所有的资源将不会被释放,并且你还需要自己删除这个线程对象(尽管这个对象只能使用一次).而分离线程则具有"点火以后就忘掉"的特性,你只需要启动它,它将自己自己中止和释放.

当然,这也意味者分离线程必须在堆上创建,因为它将在结束的时候调用delte(this).联合线程既可以在栈上创建也可以在堆上创建,不同通常也都是在堆上创建的.不要创建全局的线程对象,因为他们将在他们的构造函数执行的时候分配内存,这将导致内存检测系统出现一些问题.

指定栈大小

你可以在调用线程的Create函数的时候指定它的栈大小,默认的0代表使用操作系统默认的大小.

指定优先级

某些操作系统允许应用程序自己提供一个线程的优先级(时间片的大小),你可以通过wxThread::SetPriority函数来达到这个目的.优先级的数值在0到100之间,0为最低优先级而100为最高优先级,不过最好使用预定义的宏wxTHREAD*MIN_PRIORITY, wxTHREAD_DEFAULT_PRIORITY*和*wxTHREAD_MAX* PRIORITY,他们的值分别为0,50和100.SetPriority函数应该在调用Create之后,调用Run函数之前被调用.

启动线程

调用Create函数以后,线程还没有开始运行,你还需要调用wxThread::Run函数来启动线程,这个函数将会调用你自定义的Entry函数.

怎样暂停线程以等待一个外部条件

如果线程需要等待某些事情发生,你应该避免直接使用查询和什么事都不做这样的循环,这会让你的程序"忙于等待",白白浪费CPU的时间.

如果你需要等待几秒钟,你可以使用wxThread::Sleep函数.

如果你正在等待什么事情发生,你应该使用一种调用来阻止当前线程执行直到你收到事情已经发生的通知.例如,如果你在线程中使用了 socket,你应该阻塞在socket的系统调用上,直至socket收到数据,这就不会白白浪费CPU了,或者如果你正在等待一个联合线程使用的数据,你可以调用Wait其函数来阻塞自己.

有时候你可能会想用线程的Pause和Resume函数来临时将你的线程置入睡眠状态,但是这样做有两个问题,首先,暂停机制不是所有的系统都支持,有些系统(尤其是POSIX标准的系统)的Pause是模拟的,线程必须调用TestDestroy并且在这个函数返回True的时候立刻中断自己的运行.第二个问题是,调用Pause的线程有时候很难回复正常运行,因为这使得操作系统可能在任何时候中止你的线程的运行,如果你的线程正在锁定一个信号量,很可能来不及释放这个信号量导致出现死锁.

因此,使用Pause和Resume并不是一个很好的设计,你应该尽可能使用信号量或者关键区域(参见下一小节)来重新设计你的代码.

线程中止

正如前面我们谈到的那样,分离线程是自动结束和释放自己的.而对于联合线程,你可以简单的调用wxThread::Wait函数.或者在一个GUI程序中,你可以在系统空闲事件处理函数中调用wxThread::IsAlive函数,然后仅在IsAlive返回False的时候调用 Wait函数.Wait函数是释放联合线程资源的唯一的方法.

你可以调用wxThread::Delete来请求删除一个线程,不过要让它工作正常,你需要在你的线程中周期性的调用TestDestroy函数(译者注:根据经验,在Windows平台上,对于联合线程使用这种方法好像不是一个好主意).

17.3 用于线程同步的对象

在几乎所有的线程使用中,数据都是几个线程共享的.当有两个或以上的线程试图访问同一个数据的时候,无论这个数据是一个对象还是一个资源,这种访问都应该被同步,以避免数据在同一时刻被超过1个线程访问或者修改.因为应用程序中充满了所谓的不变量,比如,对于一个链表来说,我们总认为它的第一个元素是有效的,每个元素都指向它的下一个元素,最后的一个元素是空指针.但是在对链表进行插入新元素操作的时候,有一小段时间间隔,这个所谓不变量是被打破的.这时候假如有两个线程同时在使用这个链表,如果没有进行数据同步的动作,就会出现不可知的问题.因此你必须保证,在你插入元素这一小段时间间隔内,没有别的线程在访问同样的数据.

保证所有的共享数据被各个访问它的线程快速的并且是以一个合理的顺序访问是程序员自己的责任.因此这一小节我们来介绍一下wxWidgets提供了哪些类来帮助程序员达到这个目的.

wxMutex

这个名字来源于mutual exclusion(共有的互斥量),它是最简单的一种数据同步手段.它可以保证同一个时刻只有一个线程在访问某一部分数据.要获取数据的访问权,线程必须调用wxMutex::Lock函数,这将阻塞当前线程的执行直到它所请求的数据已经不再有任何别的线程使用.而在它开始使用这个数据以后,别的线程对 wxMutex::Lock的调用同样将被阻塞,直至当前使用的线程调用wxMutex::Unlock函数释放它所使用的资源.尽管你可以直接使用 wxMutex的Lock和Unlock函数,我们还是推荐你使用wxMutexLocker类来使用wxMutex,这将确保你不会忘记在Lock以后调用Unlock函数(译者注:你可以想像假如你忘了Unlock的后果,呵呵),因为这两个函数被隐藏在wxMutexLocker类的构造函数和析构函数中,因此,即使发生了异常,wxMutexLocker类仍然会在自己被释放的时候进行Unlock.

下面的代码中,我们确信MyApp有一个wxMutex类型的变量m_mutex:

```
void MyApp::DoSomething()
{
    wxMutexLocker lock(m_mutex);
    if (lock.IsOk())
    {
        ... do something
    }
    else
    {
        ... we have not been able to
        ... acquire the mutex, fatal error
    }
}
```

使用互斥量有三个重要的规则:

1. 线程不可以锁定已经被锁定的互斥量(不允许互斥量递归). 尽管有些系统允许你这样做,但这是不可移植的.
2. 线程不允许解锁别的线程锁定的互斥量. 如果你需要这个功能,参考我们马上会讲到的信号

量机制.

3. 如果你的线程即使无法锁定互斥量也还有别的事情可以做,你应该先使用wxMutex::TryLock函数判断是否可以锁定.这个函数是立即返回的,返回值为可以锁定(wxMUTEX_NO_ERROR)或者不可锁定(wxMUTEX_DEAD_LOCK或wxMUTEX_BUSY). 这在主线程中尤其有用,因为主线程(GUI线程)是不可以被阻塞的,否则它将不能响应任何用户的输入.

死锁

如果两个线程在互相等待对方已经锁定的互斥量,我们称之为发生了死锁.举例来说,假如线程A已经锁定了互斥量1,线程B已经锁定了互斥量 2,线程A正等待锁定互斥量2,而线程B正等待锁定互斥量1,那么,他们两个人将无限期的等待下去,在某些系统上,如果出现这种情况,Lock或者Unlock或者TryLock函数将返回错误码wxMUTEX_DEAD_LOCK,但是在另外一些系统上,除非你把整个程序杀死,否则他们将一直等下去.

解决死锁的方法有一下两种:

- 修改顺序.一个一致的互斥量锁定顺序将减少死锁发生的概率.在前面的例子中,如果线程A和线程B都要求先锁定互斥量1再锁定互斥量2,则死锁将不会发生.
- 使用TryLock. 在成功锁定第一个互斥量以后,在后续的互斥量锁定之前都使用TryLock函数判断,如果TryLock返回失败,解锁第一个然后重新开始锁定第一个. 这种方法系统开销较大,但是如果修改顺序的方法有明显的缺陷或者导致你的代码乱七八糟,你可以考虑使用这种方法.

wxCriticalSection

关键区域用来保证某一段代码在某一个时刻只被一个线程执行,而前面介绍的互斥量则用来保证互斥量在某一个时刻只被一个线程锁定.他们之间是非常相似的,除了在某些系统上,互斥量是系统范围内的变量而关键区域只在本应用程序范围内有效.在这样的系统上,使用关键区域的效率会比使用互斥量高一点点.也因为这些细微的差别,他们的一些术语也略有不同,互斥量称为锁定(或者装载)和解锁(或者卸载),而关键区域称为进入或者离开.

关键区域也有对应的wxCriticalSectionLocker对象,出于和wxMutexLocker同样的原因,你应该尽量使用它而不要直接使用wxCriticalSection的函数.

wxCondition

所谓条件变量wxCondition,是用来指示共享数据的某些条件已经满足.比如,你可以使用它来指示一个消息队列已经有数据到来.而共享数据本身(在这里指的这个消息队列)通常还需要另外使用一个互斥量来保护.

你可以通过锁定互斥量,检测队列有无数据,然后释放信号量这样的循环来进行消息队列数据的处理,不过如果队列里一直没有数据,这样的作法也太浪费了,时间全部浪费在锁定和解锁互斥量上面了.象这种情况,最好是使用条件变量,这样消息处理线程就可以被阻塞直到等到别的线程把事件放入事件队列以后发出通知事件.

多个线程可能都在等待同一个条件,这时你可以选择唤醒一个线程还是唤醒多个线程,唤醒一个线程的函数是Signal,唤醒所有正在等待的线程的函数是Broadcast.如果有多个条件都是由同一个wxCondition通知的,你必须使用Broadcast函数,否则可能某个线程被唤醒了但是却什么也做不了,因为它的条件还没有满足,而另外的可以满足条件的那个线程却无法唤醒了.

wxCondition使用举例

我们来假设一下我们有两个线程:

一个是生产线程,它负责产生10个元素并且将其放入队列,然后发送队列满信号并且在继续填充元素之前等待队列空信号.

一个是消费线程,它在收到队列满信号的时候移除队列中所有的元素.

我们需要一个互斥量m_mutex,用来保护整个队列和两个条件变量:m_isFull和m_isEmpty.这个互斥量被传递给两个条件变量的构造函数作为参数.另外你需要总是显示判断条件是否满足,然后再开始等待通知,因为可能在你还没有开始等待之前,已经有一个信号通知了,由于你还没有等待,那个信号就丢失了.

我们来看看生产线程的Entry函数的伪代码:

```
while ( notDone )
{
   wxMutexLocker lock(m_mutex) ;
   while( m_queue.GetCount() &gt; 0 )
   {
      m_isEmpty.Wait() ;
   }
   for ( int i = 0 ; i &lt; 10 ; ++i )
   {
      m_queue.Append( wxString::Format(wxT("Element %d"),i) ) ;
   }
   m_isFull.Signal();
}
```

消费线程:

```
while ( notDone )
{
   wxMutexLocker lock(m_mutex) ;
   while( m_queue.GetCount() == 0 )
   {
      m_isFull.Wait() ;
   }
   for ( int i = queue.GetCount() ; i &gt; 0 ; i )
   {
      m_queue.RemoveAt( i ) ;
   }
   m_isEmpty.Signal();
}
```

Wait函数首先Unlock其内部的互斥量 ,然后等待条件被通知.当它被通知唤醒时,会首先再次锁定内部的信号量,因此数据同步时非常严格满足的.

另外,在Wait函数被唤醒之后再次检测条件是否满足也是必要的,因为在信号被发送和线程被唤醒之间可能发生某些事情,导致条件又一次不满足了;另外,系统有时候也会产生一些假的信号导致Wait函数返回.

Signal可能在Wait之前发生,正象pthread中的那样,这时这个信号会丢失.因此如果你想要确定你没有错过任何信号,你必须保证和条件变量绑定的互斥量在最开始就处于锁定状态,并且在你调用Signal函数之前再次尝试锁定它,这意味着对Signal的调用将被阻塞直到另外一个线程调用了Wait函数.

OK,上面的这段话读起来比较费劲,我们来看一个例子,在这个例子中,主线程创建了一个工作线程,工作线程的Signal函数直到主线程调用了Wait以后才能被调用:

```
class MySignallingThread : public wxThread
{
public:
    MySignallingThread(wxMutex *mutex, wxCondition *condition)
    {
        m_mutex = mutex;
        m_condition = condition;
        Create();
    }
    virtual ExitCode Entry()
    {
        ... do our job ...
        // 告诉其它线程我们马上就要退出了.
        // 我们必须先锁定信号量,这个动作会阻塞自己
        // 直到主线程调用了Wait
        wxMutexLocker lock(m_mutex);
        m_condition.Broadcast(); // 我们只有一个线程在等待,所以等同于Signal()
        return 0;
    }
private:
    wxCondition *m_condition;
    wxMutex *m_mutex;
};
void TestThread()
{
    wxMutex mutex;
    wxCondition condition(mutex);
    // 互斥量应该先出于锁定状态
    mutex.Lock();
    // 先创建和运行工作线程,注意这个线程不能退出
    // 除非我们解锁了互斥量
    MySignallingThread *thread =
        new MySignallingThread(&mutex, &condition);
    thread->Run();
    // Wait工作线程退出,Wait函数将自动解锁和它绑定的互斥量
    // 因此工作线程可以继续直至发出Signal并且终至自己.
    condition.Wait();
    // 我们收到了Signal就可以退出了.
}
```

当然上面的这个例子指示出于演示如何实现条件变量中第一个Singal在第一个Wait之后执行,如果单就代码例子实现的功能来说,我们应该直接使用一个联合线程,然后在主线程调用wxThread::Join函数就可以了.

wxSemaphore

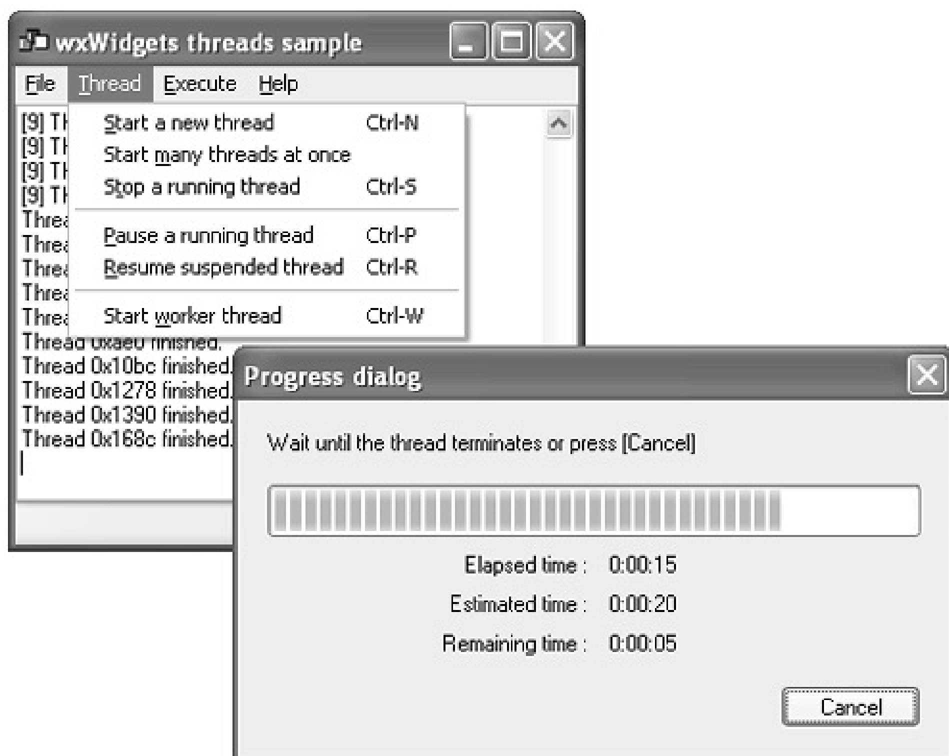
信号量(wxSemaphore)可以通俗的看成一个互斥量和一个记数器的结合,它和记数器最大的不同在于信号量的值可以被任何线程更改,而不仅仅是拥有它的那个线程.所以你也可以把信号量看作是一个没有主人的记数器.

如果一个线程调用信号量的Wait函数,这个调用将阻塞,除非记数器当前为一个正数,然后Wait函数将记数器减一,然后返回.而对Post函数的调用则将增加记数器的值然后返回.

wxWidgets实现的信号量还有一个额外的特性,你可以在其构造函数中指定一个记数器的最大值,默认为0表明最大值没有限制,如果你给定了一个最大值,而Post函数的调用使得当前的记数器超过了这个最大值,你将会得到一个wxSEMA_OVERFLOW错误.让我们再回到前面说的用信号量实现特殊互斥的描述:

- 一个可以被不同的线程锁定和解锁的互斥量可以通过一个记数器最大值为1的信号量实现,互斥量的Lock函数等同于信号量的Wait函数而互斥量的Unlock函数等同于信号量的Post函数.
- 前一个线程调用Lock(Wait)发现是一个整数值,于是减一,然后立即继续.
- 第二个线程调用Lock发现是零,将必须等待某个线程(不一定是前一个线程)调用Unlock(Post).

你可以在wxWidgets自带的samples/thread中找到一个用来演示多线程编程的例子.如下图所示.在这个例子中,你可以启动,停止,暂停,恢复线程的运行.它演示了一个工作线程周期性的通过wxPostEvent往主程序发送事件,一个进度条对话框用来指示当前进度并在进度到达最后的时候取消工作线程的运行.



17.4 多线程的替代方案

如果线程使用的复杂性让你感到气馁,也许你可以尝试一些简单的替代方案,比如使用定时器,空闲时间处理或者两者一起使用.

使用wxTimer

wxTimer类让你的程序可以周期性的收到提示,或者在某个特定的时间间隔收到提醒.如果你要使用线程处理的事情可以分成小的时间片,每隔几个毫秒处理一次,以便你的应用程序可以有足够的时间响应用户的输入,你就可以使用wxTimer来代替多线程.

你可以自己选择提醒的通知方式,如果你更喜欢使用虚函数,就实现一个wxTimer的派生类,然后重载其Notify函数,如果你更倾向使用事件机制,就给你的wxTimer构造函数指定一个wxEvtHandler指针(或者使用SetOwner)函数,然后使用EVT_TIMER(id, func)事件映射宏来将事件映射到对应的处理函数.

你可以给wxTimer的构造函数或者SetOwner函数传递一个可选的定时器标识符,这个标识符可以用在EVT_TIMER事件映射宏中.这在你需要使用多个定时器的时候比较有用.

使用Start函数启动定时器,需要传递的参数包括一个毫秒为单位的定时器时长和可选的wxTIMER_ONE_SHOT指示(如果你只希望收到一次提示).Stop函数用来终止某个定时器,IsRunning函数用来检测当前定时器是否出于运行状态.

下面的代码演示了怎样通过事件的方式使用定时器:

```
#define TIMER_ID 1000
class MyFrame : public wxFrame
{
public:
    ...
    void OnTimer(wxTimerEvent& event);
private:
    wxTimer m_timer;
};
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_TIMER(TIMER_ID, MyFrame::OnTimer)
END_EVENT_TABLE()
MyFrame::MyFrame()
       : m_timer(this, TIMER_ID)
{
    // 1秒的间隔
    m_timer.Start(1000);
}
void MyFrame::OnTimer(wxTimerEvent& event)
{
    // 你可以在这里作任何你希望1秒执行一次的动作
}
```

注意这里的时间间隔很难作到精确,实际的间隔时长要看定时器时间处理函数执行之前发生的事情的忙闲状况而定.

当我们想精确的测量时长的时候,wxStopWatch是一个很有用的类,它的构造函数开始记录时间,你可以暂停和恢复它以便获得某个特定时长的精确时间.

```
wxStopWatch sw;
SlowBoringFunction();
// 暂停监视
sw.Pause();
wxLogMessage("The slow boring function took %ldms to execute",
             sw.Time());
// 恢复监视
sw.Resume();
SlowBoringFunction();
wxLogMessage("And calling it twice took %ldms in all", sw.Time());
```

空闲时间处理

另外一种代替多线程处理的方法是使用空闲时间处理函数.应用程序类和所有的窗口类在所有其它的事件处理完以后都会收到系统空闲事件,你可以拦截这个事件并增加自己的处理函数.如果你希望收到更多的空闲事件,你可以调用wxIdleEvent::RequestMore函数,否则一次空闲事件处理完成以后,要等到下次用户界面事件处理周期结束才会再次收到系统空闲事件.另外,通常你还需要调用wxIdleEvent::Skip函数,以便别的对象的系统空闲事件处理函数可以被执行.

在下面的例子中,假想的函数FinishedIdleTask在每次空闲事件处理函数中处理一部分工作,这个函数在整个工作完成以后返回True.

```
class MyFrame : public wxFrame
{
public:
    ...
    void OnIdle(wxIdleEvent& event);
    // 作一小部分工作,如果做完则返回True
    bool FinishedIdleTask();
};
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_IDLE(MyFrame::OnIdle)
END_EVENT_TABLE()
void MyFrame::OnIdle(wxIdleEvent& event)
{
    // 进行空闲事件处理,如果没有处理完
    // 再次请求空闲事件
    if (!FinishedIdleTask())
        event.RequestMore();
    event.Skip();
}
```

虽然我们的例子中使用的是frame窗口,并不意味着空闲事件只能存在于顶层窗口,实际上,任何窗口都可以处理这个事件.比如,如果你想制作一个定制的图像显示控件,如果窗口大小发生了变化,这个控件只在空闲事件才进行重画以避免窗口大小改变时立即重画可能引发的闪铄.为了确定这个空闲处理不被应用程序的空闲事件处理函数打断,你可以在你的自定义控件中重载虚函数OnInternalIdle,并调用其基类的OnInternalIdle 函数.它的代码大致象下面的样子:

```
void wxImageCtrl::OnInternalIdle()
{
    wxControl::OnInternalIdle();

    if (m_needResize)
    {
        m_needResize = false;
        SizeContent();
    }
}
void wxImageCtrl::OnSize(wxSizeEvent& event)
{
    m_needResize = true;
}
```

有时候你可能想强制执行系统空闲处理函数,即使没有任何事件导致系统空闲事件被再次调用.你可以通过wxWakeUpIdle函数强制启动空闲事件处理.另外一个方法是启动一个不做任何事情的定时器,由于定时器会定时送出定时器事件,因此在定时器事件处理完以后(实际上什么事也没做),就会周期性的引发系统空闲事件处理,要强制立即处理所有的空闲事件,你可以直接调用wxApp::ProcessIdle函数,但是依平台的不同,这可能会影响到内部空闲事件处理(比如在GTK+平台上,窗口重绘都是在空闲时间完成的).

使用空闲事件进行用户界面更新的内容,我们已经在第9章,"创建定制的对话框"中介绍过,它用来另外一种形式的系统空闲事件wxUpdateUIEvent.

强制处理用户界面更新事件

当一个应用程序忙于处理一个很耗时的工作的时候,它可能来不及更新用户界面,这时候用户界面就好像被冻结一样,要避免这种情况,你可以在那个耗时的任务中周期性的调用wxApp::Yield函数(或者它的等价体wxYield).不过这种方法应该尽量少的使用,因为它可能会导致不可知的问题.比如,Yield可能会导致处理用户命令事件,这个事件可能会导致特定的任务被重新执行,尽管我们目前仍在执行这个任务.我们称之为重入.函数 wxSafeYield会首先禁用所有的窗口,然后Yield,然后再重新使能所有的窗口以尽可能避免重入.如果你给wxApp::Yield函数传递 True作为参数,那么如果正在进行Yield,第二次的Yield动作将放弃,这也是另外一种降低重入风险的办法.

如果你希望周期的更新特定显示,直接调用wxWindow::Update函数会比较好,因为这将只会导致未处理的重绘事件被处理.

第十七章小结

正如给一个银行业务员的业务窗口从一个变成两个不大会提高他每小时接待的顾客数目一样,多线程并不会让你的程序运行的更快(至少在普通的硬件环境下是这样).然而,它可以让你的程序看上去比不用线程更流畅了,正如那个业务员在一个柜台等刷卡的时候可以跑到另外一个窗口继续办公一样,多线程可以让你的系统的资源使用的更有效率.在解决某些特定的问题方面,它也比不用多线程的手段看上去更优雅.在这一章的最后,我们还介绍了多线程替代方案相关的内容,包括空闲事件处理,定时器和Yield等.

关于多线程编程还有很多方面的问题这里没有涉及到,如果你对更深入的内容感兴趣,推荐你阅读David R. Butenhof写的书"Programming with POSIX Threads".

下一章我们来看看怎样使用socket编程以便在进程间传递数据.

第十八章使用wxSocket编程

socket是一个数据传输的管道.socket并不关心它正在传输什么类型的数据,也不关心数据从和而来,或者说要到哪里去.它的任务就是把数据从A点传输到B点.每次你访问web,收发email,登录你的即时消息帐号等等时候,你在都使用着socket.socket可以被用来再任何支持 socket的设备之间建立连接,包括连接一台电脑和一台电冰箱(只要它支持socket).

socket编程的API最初是BSD unix系统的一部分,因为其起源的单一性,这个API变成了一种标准.所有现代的操作系统都会实现一个socket层,来提供按照TCP或者UDP协议通过网络(比如国际互联网)向外发送数据.使用wxWidgets提供的wxSocket,你可以安全的从一台电脑向另外一台电脑发送任何数量的数据.本章也将涉及一些socket技术的基础知识,但是socket操作本身是非常简单明了的.

虽然基本的socket操作是非常简单的,在Windows,Linux和Mac OSX平台上也是非常类似的,但是每个平台在实现socket的时候还是有一些细微的差别,必须针对某个特定的平台作一些适配.基于事件的socket操作在各个平台上的差异就更为突出,这使得在各个平台上使用这种机制都成为一个挑战.而wxWidgets则使用wxSocket类屏蔽了这些差别,从而使得制作基于事件的跨平台的socket程序变得相对容易.

另外需要注意的是,到作者停笔前为止,wxWidgets还不支持UDP协议的数据收发,也许在将来的版本中会增加UDP的支持.

18.1 Socket类和功能概览

socket操作的核心类是wxSocketBase,它提供了类似发送和接收数据,关闭连接,错误报告等这样的功能.创建一个监听socket或者连接到一个socket服务器,你需要分别使用wxSocketServer和wxSocketClient.wxSocketEvent用来通知应用程序 socket上有事件发生.虚类wxSocketBase和它的一些子类比如wxIPV4address让你可以指定特定的远端地址和端口.最后, wxSocketInputStream和wxSocketOutputStream等这些流对象让你以流的方式处理socket上的数据移动和传输.关于流操作的更多内容参见第14章,"文件和流操作"

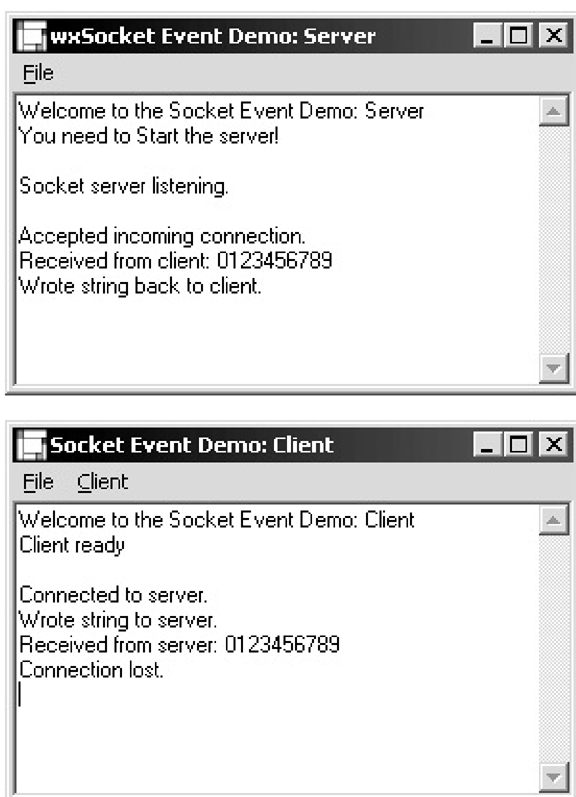
正如我们在稍后的"Socket标记"小节中即将讨论的那样,socket可以以不同的方式使用.传统的使用线程的操作方式将禁止 socket事件的产生和发送,而在线程中以阻塞的方式进行socket的操作.而另一方面,你也可能使用基于事件的方式以便逃避使用线程的复杂性. wxWdigets将在需要的时候通过事件通知你需要对某个socket进行操作了.通过这种方式,数据的接收是放在后台的,你仅需要在有数据到来的时候处理它,它将不会阻塞你的GUI界面,也没有基于每个线程一个socket的实现的那种复杂性.

本章我们通过一个完整的例子来介绍wxSocket的这两种使用方法以及使用到的那些wxSocket类的API.虽然仅仅是一个例子,但是例子中的代码都可以作为正式的代码来使用.

18.2 Socket及其基本处理介绍

让我们直接开始一个基于事件的socket客户机和服务器的例子,作为对wxWidgets中socket编程的介绍.代码是相当直观的,只需要你有一点最基础的socket编程的背景.为了简洁起见,所有GUI操作的部分将被省略,我们只关注那些Socket有关的函数.完整的代码可以在光盘的examples/chap18目录中找到.例子中用到的socket API都附有详细的使用手册.

这个例子程序的功能是很简单的,服务器倾听连接请求,当有客户端建立连接的时候,服务器首先从socket上接收10个字符,然后再把这10个字符发送回去.相应的,客户端在建立连接以后先发送10个字符,然后等待接收10个响应字符.在例子中,这10个字符写死为 "0123456789".服务器端和客户端的程序运行的样子如下图所示:



客户端的代码

下面列出了客户端的关键代码

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(CLIENT_CONNECT, MyFrame::OnConnectToServer)
    EVT_SOCKET(SOCKET_ID,    MyFrame::OnSocketEvent)
END_EVENT_TABLE()
void MyFrame::OnConnectToServer(wxCommandEvent& WXUNUSED(event))
{
    wxIPV4address addr;
    addr.Hostname(wxT("localhost"));
    addr.Service(3000);
    // 创建Socket
    wxSocketClient* Socket = new wxSocketClient();
    // 设置要监视的Socket事件
    Socket->SetEventHandler(*this, SOCKET_ID);
    Socket->SetNotify(wxSOCKET_CONNECTION_FLAG |
                                    wxSOCKET_INPUT_FLAG |
                                    wxSOCKET_LOST_FLAG);
    Socket->Notify(true);
    // 等待连接事件
    Socket->Connect(addr, false);
}
void MyFrame::OnSocketEvent(wxSocketEvent& event)
{
    // 从事件获取socket
    wxSocketBase* sock = event.GetSocket();
    // 所有事件共享的一块缓冲(Common buffer shared by the events)
    char buf[10];
    switch(event.GetSocketEvent())
    {
        case wxSOCKET_CONNECTION:
        {
            // 填充'0'-'9'的ASCII码
            char mychar = '0';
            for (int i = 0; i &lt; 10; i++)
            {
                buf[i] = mychar++;
            }
            // 发送10个字符到对端
            sock->Write(buf, sizeof(buf));
            break;
        }
        case wxSOCKET_INPUT:
        {
            sock->Read(buf, sizeof(buf));
            break;
        }
        // 服务器在发送10个字节以后关闭了连接
        case wxSOCKET_LOST:
        {
            sock->Destroy();
            break;
        }
    }
}
```

服务器端代码

下面列出了服务器端的代码

```
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU(SERVER_START, MyFrame::OnServerStart)
    EVT_SOCKET(SERVER_ID,  MyFrame::OnServerEvent)
    EVT_SOCKET(SOCKET_ID,  MyFrame::OnSocketEvent)
END_EVENT_TABLE()
void MyFrame::OnServerStart(wxCommandEvent& WXUNUSED(event))
{
    // 创建地址,默认为localhost:0
    wxIPV4address addr;
    addr.Service(3000);
    // 创建一个Socket,保留其地址以便我们可以在需要的时候关闭它.
    m_server = new wxSocketServer(addr);
    // 检查Ok函数以判断服务器是否正常启动
    if (! m_server->Ok())
    {
        return;
    }
    // 设置我们需要监视的事件
    m_server->SetEventHandler(*this, SERVER_ID);
    m_server->SetNotify(wxSOCKET_CONNECTION_FLAG);
    m_server->Notify(true);
}
void MyFrame::OnServerEvent(wxSocketEvent& WXUNUSED(event))
{
    // 接受连接请求,并创建Socket
    wxSocketBase* sock = m_server->Accept(false);
    // 告诉这个新的Socket它的事件应该被谁处理
    sock->SetEventHandler(*this, SOCKET_ID);
    sock->SetNotify(wxSOCKET_INPUT_FLAG | wxSOCKET_LOST_FLAG);
    sock->Notify(true);
}
void MyFrame::OnSocketEvent(wxSocketEvent& event)
{
    wxSocketBase *sock = event.GetSocket();
    // 处理事件
    switch(event.GetSocketEvent())
    {
        case wxSOCKET_INPUT:
        {
            char buf[10];
            // 读数据
            sock->Read(buf, sizeof(buf));
            // 写回数据
            sock->Write(buf, sizeof(buf));
            // 服务器接受的这个socket已经完成任务,释放它
            sock->Destroy();
            break;
        }
        case wxSOCKET_LOST:
        {
            sock->Destroy();
            break;
        }
    }
}
```

连接服务器

这一小段我们来解释一下怎样创建一个客户端Socket并且用它连接某个Server.

Socket地址

所有的socket地址相关的类都是基于虚类wxSockAddress,它提供了基于socket标准的所有地址相关的参数和操作.而 wxIPV4address类则具体实现了当前应用最广泛的标准国际地址方案IPV4.wxIPV6address类是用来提供IPv6支持的,不过它的功能实现的并不完整,等到IPv6在全

世界范围内广泛使用的那天,这个类当然会相应的变得完整.

注意:如果地址使用的是一个长整型,那么它期待的是网络序排列方式,它返回的长整型地址也总是网络序排列方式.网络序是对应的是Big endian(Intel和AMD的x86体系使用的是little endian,而Apple的系统使用的是big endian).你可以使用字节序转换宏wxINT32_SWAP_ON_LE来进行平台无关的字节续转换,这个宏只在使用little endian的平台上才进行相应的转换工作.如下所示:

```
IPV4addr.Hostname(wxINT32_SWAP_ON_LE(longAddress));
```

Hostname可以采用的参数包括一个wxString类型的字符串(比如 www.widgets.org)或者一个长整型的IP地址(前面已经提到过,采用big endian),如果没有任何参数,则Hostname返回当前主机的主机名.

Service用来设置远端端口,你可以指定一个wxString类型的已知服务名或者直接指定一个short类型的整数.如果不带任何参数,Service返回当前指定的远端端口.

IPAddress函数返回一个十进制的以点分割的wxString类型的远端ip地址.

AnyAddress将地址设置为本机的任何IP地址,相当于将地址设置为INADDR_ANY.

Socket客户端

wxSocketClient继承自wxSocketBase并且同时继承了所有的通用Socket操作函数.新增的少数几个函数主要用来发起和建立远端连接.

Connect函数采用一个wxSockAddress参数以便知道要连接的远端地址和端口.正如前面提到的那样,你应该使用类似 wxIPV4address这样的地址而不能直接使用wxSockAddress.第二个参数是一个bool类型,默认为true,指示是否应该等连接建立再返回.如果这个函数在主线程中运行,所有的GUI都将冻结直至这个函数返回.

WaitOnConnect用来在Connect被以false作为第二个参数调用以后(不阻塞)调用.第一个参数指示要等待的秒数, 第二个参数则用来指示毫秒数.无论连接函数成功还是失败,这个函数都将返回成功.只有当连接函数返回超时的时候,这个函数才会返回失败.如果第一个参数是 -1,则代表使用默认的超时时长,通常是10分钟.也可以使用SetTimeout函数修改默认的超时时长.

Socket事件

所有的Socket事件都是使用同一个事件映射宏EVT_SOCKET指定的.

EVT_SOCKET(identifier, function)宏将标识符为identifier的事件发送给指定的函数处理.处理函数的参数类型为wxSocketEvent.

wxSocketEvent事件非常简单,内部存储了事件的标识符和对应的wxSocket对象指针,这可以避免自己保存socket指针的麻烦.

Socket事件类型

下表列出了GetSocketEvent函数可能返回的事件类型.

wxSOCKET_INPUT	指示**socket**上有数据可以接收.无论是**socket**数据缓存原本没有数据,新收到了数据,还是说原本就有数据,只是用户还没有读完,都将产生这个事件.
wxSOCKET_OUTPUT	这个事件通常在socket的Connect函数第一次连接成功或者说Accept刚刚接受了一个新的Socket的时候产生,并且通常是产生在socket的写操作失败,缓冲区的数据又从无到有的时候.
wxSOCKET_CONNECTION	对于客户端来说,用来只是Connect动作已经成功了,对于服务端来说,指示新接受了一个Socket.
wxSOCKET_LOST	用来指示接收数据时针对socket的关闭操作.这通常意味着对端已经关闭了socket.这个事件在连接失败的时候也有可能产生.

wxSocketEvent的主要成员函数

GetSocket返回指向产生这个事件的wxSocketBase对象的指针.

GetSocketEvent返回对应的上表列出的事件类型.

使用Socket事件

要处理socket事件,你需要首先指定一个事件处理器并且指定你想要处理的事件类型.wxSocketBase支持的各种事件宏,你可以在上面的服务器端例子中监听socket创建以后的代码中看到.需要注意的事,对Socket事件的设置仅对当前的socket起作用,如果你希望监听别的socket的相关事件,你需要对那个socket再次设置监听事件.

SetEventHandler函数将某个事件标识符和相应的事件处理器关联起来. 事件标识符必须和事件处理器对应的事件表中指定的标识符相对应.

SetNotify用来设置想要监听的事件,它的参数是一个bit为列表,比如wxSOCKET_INPUT_FLAG | wxSOCKET_LOST_FLAG将监听有数据到来以及socket被关闭事件.

Notify使用一个bool类型的参数,来指示你是否想或者不想收到当前指定的事件.它的作用是让你在SetNotify之后可以不带事件指示来打开或者关闭事件监听.

Socket状态和错误提醒

在讨论数据发送和接收之前,我们先来描述一下socket状态和socket的错误提醒,以便我们在讨论数据接收的时候可以引用他们.

Close函数关闭socket,禁止随后的任何数据传输并且会通知对端socket已经被关闭.注意可能在关闭之前已经缓存了一些socket事件,因此在socket被关闭之后你可能还要准备好处理可能缓存的socket事件.

Destroy函数应该代替针对socket的delete操作,原因和Window对象类似,有可能队列中仍然有针对这个socket的事件,因此,在系统事件队列处理完以后再释放这个socket是一个安全的作法,Destroy函数正是提供了这个功能.

Error函数返回True如果上次的socket操作遇到某种错误.

GetPeer返回一个wxSockAddress引用,它包含当前socket的对端信息比如IP地址和端口号.

IsConnected返回是否这个socket已经成功连接.

LastCount返回最近一次读写操作成功进行的字节数.

LastError返回最近一次的错误码.注意如果操作成功并不会更新最近一次的错误码,因此你需要使用Error函数来判断最近一次操作是否成功.socket所支持的错误码如下表所示:

wxSOCKET_INVOP	非法操作,比如使用了非法的地址类型.
wxSOCKET_IOERR	I/O错误,比如无法创建和初始化socket.
wxSOCKET_INVADDR	不正确的地址, 比如试图连接空地址或者不完整的地址.
wxSOCKET_INVSOCK	socket使用方法不正确或者尚未初始化.
wxSOCKET_NOHOST	指定的地址不存在.
wxSOCKET_INVPORT	无效端口.
wxSOCKET_WOULDBLOCK	socket被指示为非阻塞socket,但是操作将导致阻塞 (参见socket模式的讨论).
wxSOCKET_TIMEDOUT	socket操作超时.
wxSOCKET_MEMERR	socket操作时内存分配失败.

Ok返回True的条件是: 客户端Socket必须已经和Server建立连接或者服务端Socket已经成功绑定了本地地址并且开始监听客户端连接

SetTimeout指定阻塞式访问的超时时长.默认为10分钟.

发送和接收Socket数据

wxSocketBase提供了各种基本的或高级的读写socket操作.所有操作都将保存相关的数据并且支持使用LastCount返回成功操作的字节个数,LastError返回最近一次遇到的操作错误码.

接收

Discard函数删除所有的socket接收缓冲区数据.

Peek函数让你可以读取缓冲区的数据但是不将socket缓冲区清除.你必须指定要Peek的数据的大小并且自己提供Peek目的地的缓冲区.

Read函数和Peek一样,只是它在成功获取数据以后会清除相应的Socket接收缓冲区.

ReadMsg函数对应于WriteMsg函数,将会完整的接收WriteMsg发送的数据,除非需要系统错误.注意如果ReadMsg开辟的缓冲区比WriteMsg发送的数据少,则多出的数据将被直接删除.

Unread将数据放回接收缓冲区,你需要指定希望放回去的数据的字节数.

发送

Write函数以参数中数据指针指向的缓冲作为开始位置,向socket写入参数中指定的数据大小.

WriteMsg和Write的区别在于,wxWidgets会增加一个消息头,以便接收端可以准确的知道消息的大小,WriteMsg发送的数据必须由ReadMsg函数接收.

创建一个Server

wxSocketServer也只对其基类wxSocketBase增加了少数几个函数用来创建和监听连接请求.要创建一个 Server,你必须指定要监听的端口.wxSocketServer使用和wxSocketClient一样的wxIPV4address类型,只是前者不需要指定远端地址.在大多数情况下,你需要调用Ok函数来判断是否绑定和监听动作已经成功.

wxSocketServer的主要成员函数

wxSocketServer构造函数使用一个地址对象用来指定监听端口,以及一个可选的Socket标记(参见下一节"Socket Flags").

Accept函数返回一个新的socket连接或者立即返回NULL,如果没有连接请求.你可以设置可选的等待标记,如果你这样做,Accept将导致程序阻塞.

AcceptWith和Accept的功能相近,只是它提供一个额外的已存在的wxSocketBase对象(引用),并且其返回值为bool型,用来指示是否接受了一个新的连接.

WaitForAccept采用一个秒参数和一个毫秒参数以指定在某个事件范围内等待新的连接请求,如果请求发生则返回True,否则超时返回False.

处理新的连接请求事件

当监听socket检测到一个新的连接请求的时候,将产生一个相应的事件.在其事件处理函数中,你可以接受这个请求并且执行任何必要的即时处理.你需要保证连接在其生命周期内不被立即关闭,你还需要为新接受的socket指定事件处理器.注意监听的socket在被关闭之前将一直在监听,而每一个新的连接请求都会创建一个新的socket.在server的整个生命周期内,同一个监听socket可以接受成千上万个新的socket.

Socket事件概述

从程序员的观点来说,基于事件的socket处理简化了socket编程,使得他们不需要关心线程的创建和释放.这个例子没有使用线程, 但是GUI界面同样不会阻塞,因为所有的数据读取都是在确信有数据到来的时候才进行的,因此会立即返回.如果有很大量的数据需要读取,你可以将它们分为

多个小部分,然后一次读一部分并将其放入你自己的缓冲区.或者你可以使用Peek函数检查当前缓冲区的数据的数量,如果没有达到需要处理的范围,你可以什么也不做,静静等待下一次数据事件通知的到来.

在下一节,我们来看看怎样使用不同的socket标记来改变socket的行为.

18.3 Socket标记

Socket的行为可能随着创建时指定的不同而有很大的差异,下表列出了Socket可以指定的标记:

wxSOCKET_NONE	普通行为(行为和底层的**send**和**recv**函数一致).
wxSOCKET_NOWAIT	读和写操作尽可能快速的返回.
wxSOCKET_WAITALL	等待所有的读写数据完成操作,除非出现系统错误.
wxSOCKET_BLOCK	在读写数据的时候阻塞GUI界面.

如果没有指定任何标记(或者指定了wxSOCKET_NONE标记),I/O操作将在部分数据被读写的时候返回,甚至在整个数据还没有传输完的情况下也是这样.这和使用阻塞方式调用底层的recv或者send函数的效果是一样的.注意这里所说的阻塞指的是函数被阻止返回,并不意味着图形用户界面被冻结.

如果指定了wxSOCKET_NOWAIT标记,I/O将立刻返回.对于读操作,它将读取所有当前输入缓冲区拥有的数据后立刻返回,对于写操作,它将尽可能多的发送数据以后立刻返回,这取决于当前的输出缓冲区的大小,这种方式等同于使用非阻塞方式调用底层函数recv或send.同样, 这里的阻塞也指的是函数返回,而不是用户界面阻塞.

如果指定了wxSOCKET_WAITALL标记,I/O操作将在所有要求的数据被读取或者被写入以后(或者发生系统错误)才会返回. 如果需要的话,将以阻塞的方式调度底层系统函数.这相当于使用一个循环重复以阻塞的方式调用recv或者send函数以便传输所有的数据.同样,这里的阻塞也指的是阻塞底层函数而不是GUI.注意ReadMsg和WriteMsg函数将隐式使用这种方式,并且忽略你可能设置的wxSOCKET_NONE或 wxSOCKET_NOWAIT标记.

用来指示wxSOCKET_BLOCK是否在I/O操作的间隙执行Yield操作(译者注:参见前面关于线程的替代方案中的描述),如果指定了这个标记,socket在底层操作间隙将不会执行Yield动作,反之,如果没有指定这个标记,那么你要非常小心这可能产生的代码重入的问题.

总的来说:

- wxSOCKET_NONE 总是试图读取或者写入一些数据,但是不关心具体是多少数据.
- wxSOCKET_NOWAIT 只关心尽快返回,即使没有读取或写任何数据.
- wxSOCKET_WAITALL 将在所有的数据都被写入或者是读取的数据达到要求的数目的时候返回.
- wxSOCKET_BLOCK 和前面的标记没有关系,只控制否则在底层操作的间隙执行Yield动作.

wxWidget中的阻塞和非阻塞socket

在wxWidgets中的阻塞方式有双重的含义.在一般的编程中,socket阻塞意味着当前的线程被挂起直至socket操作超时或者数据操作完成.如果是主线程阻塞,则用于界面也相应阻塞.

而在wxWidgets中,有两种类型的阻塞:socket阻塞和用户界面阻塞.wxSOCKET_BLOCK标记指示在socket阻塞的时候,是否同时阻塞用户界面.你也许回问,这怎么可能呢,怎么可能作到阻塞了socket操作而不阻塞用户界面呢?这主要是因为在socket被阻塞的时候,wxWidgets还可以处理所有的事件,因为socket的底层函数处理的间隙调用了wxYield,这个函数可以处理队列中所有未处理的事件,包括用户界面相关的事件.虽然在socket操作未结束之前,代码一直在socket函数中运转,但是所有事件还是可以被有序的处理.

对于刚开始使用wxWidgets的人来说,听上去这是一个很美妙的事情.如果你是第一次使用wxWidgets进行socket编程,你可能会觉得,再也不需要使用任何单独的线程来处理socket了,你可以将socket设置为wxSOCKET_WAITALL和 wxSOCKET_BLOCK,这样你可以通过事件机制处理socket数据,而GUI也不会被阻塞,不幸的是,我必须先警告你,这种想法可能是你痛苦的开始.

让我们来假设一个服务端有两个活动连接,每一个都设置了wxSOCKET_WAITALL标记.更进一步,我们假设其中一个连接正在以一种很缓慢的速度传输一个很大量的数据.Socket 1的读缓冲区没有数据了,因此它调用了wxYield,而Socket2还有未处理的事件在队列里,这时候会发生的事情是:Socket1调用的 wxYield试图处理Socket2相关的消息,但是因为Socket2的连接很缓慢,它的事件总是结束不了,它也会调用wxYield来避免GUI阻塞,这将导致出现一个名声不太好的告警消息"wxYield called recursively"(wxYield被递归调用),在Socket2的数据传输结束之前,应用程序的堆栈将最终被wxYield的递归调用给耗尽, 因为递归调用使用这些堆栈一直没有机会释放,于是人们开始联系wxWidgets社区,报告发现了一个bug,而实际上,这应该是应用程序自己的问题而不是wxWidgets的问题.应用程序不应该以这种方式来编程,它应该避免这种情形出现,因此,恐怕wxWidgets永远没有办法改正这个问题.

另外一方面,性能也是一个问题,为了不阻止GUI,你的应用程序将不得不浪费CPU的资源.试想一下,用户界面要立即响应, Socket也要不停的监视是否有数据到来以便产生事件通知应用程序,要让两者都得到满足,wxWidgets所能做的唯一的办法就是使用循环,不停的以非阻塞的方式去用系统操作select去测试socket,然后再调用wxYield处理GUI事件.

不可用的标记组合

容我再罗嗦一句,不要天真到认为wxWidgets采用了一种神奇的socket处理机制.无论这些Socket的标记在你的第一印象中看起来是多么的诱人,你都不可能同时满足下面的这些要求:

- wxSOCKET_WAITALL
- 不阻塞GUI
- 少于100%的CPU占用率
- 单线程

你可以指定wxSOCKET_WAITALL并且也不阻塞GUI,但是这将导致100%的CPU占用率.如果你愿意付出阻塞GUI作为代价(指定 wxSOCKET_BLOCK标记),你将可以得到wxSOCKET_WAITALL和0%的CPU占用率. 或者你可以使用多线程来实现同时使用wxSOCKET_WAITALL又不阻塞GUI,并且也不用100%的CPU占用率.你也可以不用

wxSOCKET_WAITALL而使用wxSOCKET_NOWAIT以便可以既不占用100%的CPU又不堵塞GUI.总之一句话,上面的四个条件, 你总可以同时满足任意三个,但是你不可以四个条件同时满足.

这些标记是怎样影响Socket的行为的

wxSOCKET_NONE, wxSOCKET_NOWAIT和wxSOCKET_WAITALL是互斥的,你不可以同时使用他们中间的任何两个,而wxSOCKET_BLOCK和 wxSOCKET_NOWAIT的组合也是没有意义的 (如果任何函数都立即返回,怎么可能阻塞GUI呢?), 因此下面五种标记的组合是有意义的:

- wxSOCKET_NONE | wxSOCKET_BLOCK: 这种组合和标准的socket调用(recv和send)的行为相同.
- wxSOCKET_NOWAIT: 和标准的非阻塞的socket调用行为相同.
- wxSOCKET_WAITALL | wxSOCKET_BLOCK: 和普通的阻塞式socket调用的行为相同,只不过recv和send函数将被连续多次调用以便接受或者发送完整的数据.
- wxSOCKET_NONE: 和标准的socket调用行为相同,只是由于在完成系统调用之前(比如数据完整接收缓冲区数据之前)调用了wxYield,因此看上去GUI并不会阻塞.
- wxSOCKET_WAITALL: 和wxSOCKET_WAITALL | wxSOCKET_BLOCK的行为相同只不过GUI将不被阻塞.

只有最后两种情况可能出现前面介绍的wxYield函数重入的问题,不过这俩组标记也是在wxWidgets中基于事件的socket编程中最主要的两种方式(因为他们阻塞了socket但是却不阻塞GUI).使用这两组标记的时候要非常小心避免这个问题,虽然它们很强大,很有用,却也往往是错误和麻烦的根源,因为它们太容易被误解了.

标准socket和wxSocket

使用wxSOCKET_NONE | wxSOCKET_BLOCK或wxSOCKET_NOWAIT的时候和直接使用socket系统调用的效果并没有不同,唯一的不同是你使用的是 wxWidgets提供的API而不是标准C的API.不过,即使这样,还是有足够的理由要使用wxSocket,这些理由包括:wxSocket提供了一个面向对象的接口,隐藏了很多平台相关的初始化代码,还提供了一些高级的函数比如WriteMsg和ReadMsg.另外,下一节我们也将看到, wxSocket也使我们可以用流的方式来操作socket.

18.4 使用Socket流

使用wxWidgets的流,你仅使用很少的代码就可以很容易传输很大量的数据.现在,假设我们要通过socket来传输一个文件.你可能使用的方法是: 打开这个文件,将所有的内容读入内存,然后将这块内存写入到socket.这种方法对于小文件来说没什么问题,但是如果这个文件是一个很多兆的大文件,将其完整读入内存对于一个速度和内存都很小的电脑来说,显得有些不太现实.而且通常我们需要对大文件进行压缩然后才发往socket以便降低网络流量.怎么办法呢,把大文件读入内存,对其整个进行压缩,然后再一次性发送socket,这样得作法在效率和实用性方面都值得怀疑.

OK,我们来想另外一种办法,每次从文件里读入一小段数据,比如几K数据,然后将其压缩,然后发往socket,如此反复.不幸得是, 小段压缩比起整个文件一起压缩来,压缩效率是大打折扣的.因此我们需要更进一步,维护一个压缩的状态,以便后面的小段数据可以使用前面的压缩信息,也可以避免多次传递压缩头信息.可是到目前为止,你的代码已经变的很庞大了,要分段读取文件,维护压缩数据,压缩并且写入socket. wxWidgets提供了一种更简便的方法.

因为wxWidgets提供了wxSocketInputStream和wxSocketOutputStream类,通过别的流来将数据读出或者写入socket是非常方便的.因为wxWidgets提供了基于文件,字符串,文本,内存以及zlib压缩的流操作,将这些流和 socket流结合起来使用,可以实现很有趣也是很强大的socket数据操作方法.现在,回过头来看看我们刚才说的通过socket压缩传输大文件的问题,我们可能已经找到了一个更方便的途径.要发送一个文件,我们可以现将来自文件的数据流通过zlib的压缩流以后发送到socket的发送流,这样我们一下在就有了强大的支持大文件,支持压缩的,每次只需要读几K的socket文件传输方法了.而在接收端,我们同样可以使用流操作将来自socket流的数据通过zlib解压缩流发送到文件输出流,最后还原为原来的文件.所有这些可能几行代码就足够了.

我们将使用线程来处理整个过程,以便我们可以既不占用100%的CPU,又不阻塞GUI(正如上一小节讨论的那样),要知道在使用socket传输大型的数据的时候,(如果不使用多线程,)这种阻塞几乎是不可避免的.

完整的例子可以在光盘的examples/chap18目录中找到.

文件发送线程

下面的例子中演示了流对象在堆上创建,FileSendThread派生自wxThread.

```
FileSendThread::FileSendThread(wxString Filename,
                                 wxSocketBase* Socket)
{
    m_Filename = Filename;
    m_Socket = Socket;
    Create();
    Run();
}
void* FileSendThread::Entry()
{
    // 如果10秒之内我们什么数据都发送不了,就超时退出
    m_Socket->SetTimeout(10);
    // 在所有数据发送完成之前,阻塞一切非socket操作
    m_Socket->SetFlags(wxSOCKET_WAITALL | wxSOCKET_BLOCK);
    // 从特定的文件流中读入数据
    wxFileInputStream* FileInputStream =
        new wxFileInputStream(m_Filename);
    // 用来写入socket的流对象
    wxSocketOutputStream* SocketOutputStream =
        new wxSocketOutputStream(*m_Socket);
    // 我们写入的将是压缩以后的数据
    wxZlibOutputStream* ZlibOutputStream =
        new wxZlibOutputStream(*SocketOutputStream);
    // 将文件的内容写入压缩流
    ZlibOutputStream->Write(*FileInputStream);
    // 写所有的数据
    ZlibOutputStream->Sync();
    // 释放ZlibOutputStream将导致发送zlib的压缩结束标记
    delete ZlibOutputStream;
    // 释放资源
    delete SocketOutputStream;
    delete FileInputStream;
    return NULL;
}
```

文件接收线程

接收例子演示了相关流对象也可以在栈上创建.FileReceiveThread派生自wxThread.

```
FileReceiveThread::FileReceiveThread(wxString Filename,
                                    wxSocketBase* Socket)
{
    m_Filename = Filename;
    m_Socket = Socket;
    Create();
    Run();
}
void* FileReceiveThread::Entry()
{
    // 如果10秒内什么也收不到,中止接收
    m_Socket->SetTimeout(10);
    // 在我们成功接收完数据之前,阻塞一切其它的代码
    m_Socket->SetFlags(wxSOCKET_WAITALL | wxSOCKET_BLOCK);
    // 用于输出数据到文件的流对象
    wxFileOutputStream FileOutputStream(m_Filename);
    // 从socket接收数据的流对象
    wxSocketInputStream SocketInputStream(*m_Socket);
    // zlib解压缩流对象
    wxZlibInputStream ZlibInputStream(SocketInputStream);
    // 将解压缩以后的结果写入文件
    FileOutputStream.Write(ZlibInputStream);
    return NULL;
}
```

18.5 替代wxSocket

虽然wxSocket提供了很多灵活性并且被很好的集成进了wxWidgets,但是它并不是实现进程间通信的唯一方法.如果你只是想进行FTP或者 HTTP的操作,你可以直接使用wxFTP或wxHTTP,它们在内部使用了wxSocket,不过这些类是不完善的,你最好还是使用CURL,它是一个通用的库,提供了使用各种Internet协议传递文件的非常直观的API,有人已经对其进行了wxWidgets封装,名字叫做wxCURL.

wxWidgets也提供了一套高级的进程间通信机制,它使用类wxServer,wxClient和wxConnection以及基于微软的DDE(动态数据交换)协议的API.实际上,在windows上,这些类是用DDE实现的,而在其它平台上,则是用socket实现的.之所以要使用这些更高层的类,是因为它比直接使用wxSocket更方便,另外一个优点是在windows平台上,使用DDE可以和别的支持DDE的程序交换数据(别的程序不必要是使用wxWidgets制作的).它的一个缺点是在别的平台上,非wxWidgets编制的程序是不能识别这种协议的,不过,如果你只需要在wxWidgets制作的程序之间交换数据的话,它还是可以满足要求的.我们将在第20章的"单个实例还是多个实例?"小节,演示一个简单的例子.

更多信息请参考wxWidgets手册中的"Interprocess Commun-ication Overview"(进程间通信概述)小节以及wxWidgets自带的samples/ipc中的例子.你也可以参考wxWidgets自带的独立帮助显示工具中的代码,它位于utils/helpview/src目录内.

第十八章小结

我们在这一章里讨论了在wxWidgets中集成了wxSocket类,并且描述了它和C语言中的socket层的关系.为了让socket编程更容易, wxWidgets还实现了socket的流操作,以便可以容易的和别的流对象进行交互来操作数据.只要你注意我们在socket标记小节中讨论的那些可能出现误解的地方,相信你可以使用wxSocket和其它相关的类制作出稳定的可以在各种平台上交叉编译的socket程序.

在下一章里,我们来看看如何使用wxWidgets提供的文档/视图框架来简化你的应用程序设计.

第十九章使用文档/视图框架

本章来讨论一下wxWidgets提供的文档和视图框架,通过使用它,那些基于文档的应用程序的使用的代码可以大为减少.另外我们还将讨论和撤消/重做操作相关的实现,我们将介绍通过怎样的途径让这个看上去非常复杂的操作变成很自然的事情.

19.1 文档/视图基础

文档/视图框架在很多编程框架中都专门提供了支持,因为它可以很大程度的简化编写类似的程序需要的代码量.

文档/视图框架主要是让你用文档和视图两个概念来建模.所谓文档,指的是那些用来存储数据和提供用户界面无关的操作的类,而视图,指的是用来显示数据的那些类.这和所谓的MVC模型(模型-视图-控制器)很相似,只不过这里把视图和控制器合在一起,作为一个概念.

基于这个框架,wxWidgets可以提供大量的用户界面控件和默认行为.你需要先定义自己的派生类以及它们之间的关系,框架本身则负责显示文件选择,打开和关闭文件,询问用户保存数据,将菜单项和对应的代码关联,甚至一些基本的打印和预览功能,还有就是重做/撤消功能的支持等.这个框架已经被高度的模块化了,允许你的应用程序通过重载和替换函数和对象的方式来更改这些默认的行为.

如果你觉得框架适合你即将制作的程序,你可以采用下面的步骤来使用这个框架.这些步骤的顺序并不是非常的严格的,你大可以先创建你的文档类,然后再考虑你的文档在应用程序中的表现形式.

1. 决定你要使用的用户界面: 微软的MDI (多文档界面,所有的子文档窗口被包含在一个父窗口内), SDI (单文档界面,每个文档一个单独的frame窗口), 或者是单一界面(同时只能打开一个文档,就象windows的写字板程序那样).
2. 基于前面的选择来使用对应的父窗口和子窗口类,比如wxDocParentFrame和wxDocChildFrame类. 在OnInit函数中创建一个父窗口的实例,对应于每个文档视图创建一个子窗口的实例(如果不是单文档界面的话).使用标准的菜单标识符创建菜单(比如wxID_OPEN和wxID_PRINT).
3. 定义你自己的文档和视图类,重载尽可能少的成员函数用于输入和输出,绘画以及初始化.如果你需要重做/撤消的支持,你应该尽早实现它而不要等到程序快完成的时候再回来返工.
4. 定义任意的子窗口(比如一个滚动窗口)用来显示视图.你可能需要将它的一些事件传递给视图或者文档类处理,比如通常它的重绘事件都需要传递给wxView::OnDraw函数.
5. 在你的wxApp::OnInit函数的开始部分创建一个wxDocManager实例以及足够多的wxDocTemplate实例,以便定义文档和视图之间的关系.对于简单的应用程序来说,一个wxDocTemplate的实例就可以了.

我们将用一个简单的叫做Doodle(参见下图)的程序来演示上面的步骤.正如它的名字那样,它支持在一个窗口上任意乱画,并且支持将这些涂鸦保存在文件里或者从文件里读取.也支持简单的重做和撤消操作.



第一步:选择用户界面类型

传统上,windows平台的多文档程序都使用的是多文档界面,我们已经在第4章,"窗口基础"中的"wxMDIParentFrame"小节对此有过描述.多文档界面使用一个父frame窗口管理和包含多个文档子frame窗口,而其菜单条则用来反应当前活动窗口或者父窗口(如果没有当前活动窗口的话) 相关联的菜单命令.

或者你也可以选择使用一个主窗口,多个顶层的用于显示文档的frame窗口的方式,这种方式下文档窗口可以不受主窗口的限制,在桌面上任意移动.这通常是Mac OS采用的风格,不过在Mac OS上,每次只能显示一个菜单条(当前活动窗口的菜单条).Mac OS上另外一个和别的平台不同的地方在于,Mac 用户并不期望应用程序的所有的窗口被关闭以后退出应用程序.Mac系统有一个应用程序菜单条,上面显示了的应用程序所有的窗口都隐藏时候可以的少数的几个命令,在wxWidgets上,要实现这种行为,你需要创建一个不可见的frame窗口,它的菜单条将在所有其它可见窗口被释放以后自动显示在那个位置.

这种技术的另外一种用法是显示一个主窗口之外的非文档视图的frame窗口.不过,这种用法非常罕见,一般都不会这样使用.另外一种方法是仅显示文档窗口,不显示主窗口,仅在最后一个文档窗口被关闭的时候显示主窗口以用来创建或者打开新的文档窗口,这种模型被近期的Microsoft Word采用,这其实是一个和Mac OS很接近的作法,只不过在Mac OS上,在这种情况下,没有任何可见的窗口,只有一个菜单条.

也许最简单的模型是只有一个主窗口,没有独立的子窗口,每次也只能打开一个文档:微软的写字板就是这样的一个例子.这也是我们的Doodle例子所选择的形式.

最后,你当然也可以创建自己的模型,或者你可以采用上面这些模型的组合方式.比如DialogBlocks就是这样的一个例子,它组合了几种方式以便用户自己作出选择.DialogBlocks中最常用的是方式是每次只显示一个视图,,当你在工程树中选择了一个文档的时候,当前视图隐藏, 新的视图打开.你也可以打开多页面支持,以便快速的在你最感兴趣的几个文档之间切换.另外,你还可以通过拖拽标题栏的方式,将某个文档以单独的窗口拖动到桌面上,以便你可以同时看到几个文档的视图.在DialogBlocks程序内部,它自己管理视图和文档以及窗口之间的关系,采用的就是和标准的 wxWidgets不同的方式.很明显,创建这样的定制文档视图管理系统需要很多时间,因此,你可能更愿意选择wxWidgets提供的标准方式.

第二步: 创建和使用frame窗口类.

对于MDI界面应用程序来说,你应该使用wxDocMDIParentFrame和wxDocMDIChildFrame窗口类,而对于主窗口和文档窗口分离的模型来说,你可以选择使用wxDocParentFrame和wxDocChildFrame类.如果你使用的是单个主窗口每次打开一个文档这种模型,你可以只使用wxDocParentFrame类.

如果你的应用程序没有主窗口,只有多个文档窗口,你既可以使用wxDocParentFrame,也可以使用 wxDocChildFrame.不过,如果你使用的是wxDocParentFrame,你需要拦截EVT_CLOSE事件,以便只删除和这个窗口绑定的文档视图,因为这个窗口类默认的EVT_CLOSE事件处理函数将删除所有文档管理器知道的视图(这将导致关闭所有的文档).

下面列出了doodle例子的窗口类定义.其中保存了一个指向doodle画布的指针和一个指向编辑菜单的指针,以便文档视图系统可以视情况更新重做和撤消菜单.

```
// 定义一个新的frame窗口类.
class DoodleFrame: public wxDocParentFrame
{
    DECLARE_CLASS(DoodleFrame)
    DECLARE_EVENT_TABLE()
public:
    DoodleFrame(wxDocManager *manager, wxFrame *frame, wxWindowID id,
        const wxString& title, const wxPoint& pos,
        const wxSize& size, long type);
    /// 显示关于对话框
    void OnAbout(wxCommandEvent& event);
    /// 获得编辑菜单指针
    wxMenu* GetEditMenu() const { return m_editMenu; }
    /// 获得画布指针
    DoodleCanvas* GetCanvas() const { return m_canvas; }
private:
    wxMenu *        m_editMenu;
    DoodleCanvas*   m_canvas;
};
```

下面的代码演示了DoodleFrame的实现.其中构造函数创建了一个菜单条和一个DoodleCanvas对象,后者拥有一个铅笔状的鼠标指针.文件菜单被传递给文档视图模型的管理对象,以便其可以增加最近使用文件的显示.

```
IMPLEMENT_CLASS(DoodleFrame, wxDocParentFrame)
BEGIN_EVENT_TABLE(DoodleFrame, wxDocParentFrame)
    EVT_MENU(DOCVIEW_ABOUT, DoodleFrame::OnAbout)
END_EVENT_TABLE()
DoodleFrame::DoodleFrame(wxDocManager *manager, wxFrame *parent,
                 wxWindowID id, const wxString& title,
                 const wxPoint& pos, const wxSize& size, long type):
wxDocParentFrame(manager, parent, id, title, pos, size, type)
{
    m_editMenu = NULL;
    m_canvas = new DoodleCanvas(this,
                    wxDefaultPosition, wxDefaultSize, 0);
    m_canvas->SetCursor(wxCursor(wxCURSOR_PENCIL));
    // 增加滚动条
    m_canvas->SetScrollbars(20, 20, 50, 50);
    m_canvas->SetBackgroundColour(*wxWHITE);
    m_canvas->ClearBackground();
    // 增加图标
    SetIcon(wxIcon(doodle_xpm));
    // 创建菜单
    wxMenu *fileMenu = new wxMenu;
    wxMenu *editMenu = (wxMenu *) NULL;
    fileMenu->Append(wxID_NEW, wxT("&New..."));
    fileMenu->Append(wxID_OPEN, wxT("&Open..."));
    fileMenu->Append(wxID_CLOSE, wxT("&Close"));
    fileMenu->Append(wxID_SAVE, wxT("&Save"));
    fileMenu->Append(wxID_SAVEAS, wxT("Save &As..."));
    fileMenu->AppendSeparator();
    fileMenu->Append(wxID_PRINT, wxT("&Print..."));
    fileMenu->Append(wxID_PRINT_SETUP, wxT("Print &Setup..."));
    fileMenu->Append(wxID_PREVIEW, wxT("Print Pre&view"));
    editMenu = new wxMenu;
    editMenu->Append(wxID_UNDO, wxT("&Undo"));
    editMenu->Append(wxID_REDO, wxT("&Redo"));
    editMenu->AppendSeparator();
    editMenu->Append(DOCVIEW_CUT, wxT("&Cut last segment"));
    m_editMenu = editMenu;
    fileMenu->AppendSeparator();
    fileMenu->Append(wxID_EXIT, wxT("E&xit"));
    wxMenu *helpMenu = new wxMenu;
    helpMenu->Append(DOCVIEW_ABOUT, wxT("&About"));
    wxMenuBar *menuBar = new wxMenuBar;
    menuBar->Append(fileMenu, wxT("&File"));
    menuBar->Append(editMenu, wxT("&Edit"));
    menuBar->Append(helpMenu, wxT("&Help"));
    // 指定菜单条
    SetMenuBar(menuBar);
    // 历史文件访问记录的显示将使用这个菜单.
    manager->FileHistoryUseMenu(fileMenu);
}
void DoodleFrame::OnAbout(wxCommandEvent& WXUNUSED(event) )
{
    (void)wxMessageBox(wxT("Doodle Sample\n(c) 2004, Julian Smart"),
                       wxT("About Doodle"));
}
```

第三步: 定义你的文档和视图类

你的文档类应该有一个默认的构造函数,而且应该使用DECLARE_DYNAMIC_CLASS和IMPLEMENT_DYNAMIC_CLASS宏来使其提供RTTI并且支持动态创建(否则你就需要重载wxDocTemplate::CreateDocument函数,以实现的文档实例创建函数).

你还需要告诉文档视图框架怎样保存和读取你的文档对象,如果你想直接使用wxWidgets流操作,你可以重载SaveObject和 LoadObject函数,就象我们例子中的作法一样.或者你可以直接重载DoSaveDocument函数和DoOpenDocument函数,这两个函数的参数为文件名而不是流对象.wxWidget流操作相关内容我们已经在第14章,"文件和流操作"中介绍过.

注意:框架本身在保存数据的时候不使用临时文件系统.这也是为什么我们有时候需要重载DoSaveDocument函数的一个理由,我们可以通过流操作将文档保存在wxTempFile中,正如我们在第14章中介绍的那样.

下面是我们的DoodleDocument类的声明部分:

```
/*
 * 代表一个Doodle文档
 */
class DoodleDocument: public wxDocument
{
    DECLARE_DYNAMIC_CLASS(DoodleDocument)
public:
    DoodleDocument() {};
    ~DoodleDocument();
    /// 保存文档
    wxOutputStream& SaveObject(wxOutputStream& stream);
    /// 读取文档
    wxInputStream& LoadObject(wxInputStream& stream);
    inline wxList& GetDoodleSegments() { return m_doodleSegments; };
private:
    wxList m_doodleSegments;
};
```

你的文档类也许要包含文档内容对应的数据.在我们的例子中,我们的数据就是一个doodle片断的列表,每一个数据片断代表从鼠标按下到鼠标释放过程中鼠标划过的所有的线段.这些片断所属的类知道怎样将自己保存在流中,这使得我们实现文档保存和读取的流操作变的相对容易.下面是用来代表这些线段片断的类的声明:

```
/*
 * 定义了一个两点之间的线段
 */
class DoodleLine: public wxObject
{
public:
    DoodleLine(wxInt32 x1 = 0, wxInt32 y1 = 0,
               wxInt32 x2 = 0, wxInt32 y2 = 0)
    { m_x1 = x1; m_y1 = y1; m_x2 = x2; m_y2 = y2; }
    wxInt32 m_x1;
    wxInt32 m_y1;
    wxInt32 m_x2;
    wxInt32 m_y2;
};
/*
 * 包含一个线段的列表,用来代表一次鼠标绘画操作
 */
class DoodleSegment: public wxObject
{
public:
    DoodleSegment(){};
    DoodleSegment(DoodleSegment& seg);
    ~DoodleSegment();
    void Draw(wxDC *dc);
    /// 保存一个片断
    wxOutputStream& SaveObject(wxOutputStream& stream);
    /// 读取一个片断
    wxInputStream& LoadObject(wxInputStream& stream);
    /// 获取片断中的线段列表
    wxList& GetLines() { return m_lines; }
private:
    wxList m_lines;
};
```

DoodleSegment类知道怎么在某个设备上下文上绘制自己,这有助于我们实现我们的doodle绘制代码.

下面的代码是这些类的实现部分:

```
/*
 * DoodleDocument
 */
IMPLEMENT_DYNAMIC_CLASS(DoodleDocument, wxDocument)
DoodleDocument::~DoodleDocument()
{
    WX_CLEAR_LIST(wxList, m_doodleSegments);
}
wxOutputStream& DoodleDocument::SaveObject(wxOutputStream& stream)
{
    wxDocument::SaveObject(stream);
    wxTextOutputStream textStream( stream );
    wxInt32 n = m_doodleSegments.GetCount();
    textStream &lt;&lt; n &lt;&lt; wxT('\n');
    wxList::compatibility_iterator node = m_doodleSegments.GetFirst();
    while (node)
    {
        DoodleSegment *segment = (DoodleSegment *)node->GetData();
        segment->SaveObject(stream);
        textStream &lt;&lt; wxT('\n');
        node = node->GetNext();
    }
    return stream;
}
wxInputStream& DoodleDocument::LoadObject(wxInputStream& stream)
{
    wxDocument::LoadObject(stream);
    wxTextInputStream textStream( stream );
```

```
    wxInt32 n = 0;
    textStream &gt;&gt; n;
    for (int i = 0; i &lt; n; i++)
    {
        DoodleSegment *segment = new DoodleSegment;
        segment->LoadObject(stream);
        m_doodleSegments.Append(segment);
    }
    return stream;
}
/*
 * DoodleSegment
 */
DoodleSegment::DoodleSegment(DoodleSegment& seg)
{
    wxList::compatibility_iterator node = seg.GetLines().GetFirst();
    while (node)
    {
        DoodleLine *line = (DoodleLine *)node->GetData();
        DoodleLine *newLine = new DoodleLine(line->m_x1, line->m_y1, line->m_x2, line->m_
        GetLines().Append(newLine);
        node = node->GetNext();
    }
}
DoodleSegment::~DoodleSegment()
{
    WX_CLEAR_LIST(wxList, m_lines);
}
wxOutputStream &DoodleSegment::SaveObject(wxOutputStream& stream)
{
    wxTextOutputStream textStream( stream );
    wxInt32 n = GetLines().GetCount();
    textStream &lt;&lt; n &lt;&lt; wxT('\n');
    wxList::compatibility_iterator node = GetLines().GetFirst();
    while (node)
    {
        DoodleLine *line = (DoodleLine *)node->GetData();
        textStream
            &lt;&lt; line->m_x1 &lt;&lt; wxT(" ")
            &lt;&lt; line->m_y1 &lt;&lt; wxT(" ")
            &lt;&lt; line->m_x2 &lt;&lt; wxT(" ")
            &lt;&lt; line->m_y2 &lt;&lt; wxT("\n");
        node = node->GetNext();
    }
    return stream;
}
wxInputStream &DoodleSegment::LoadObject(wxInputStream& stream)
{
    wxTextInputStream textStream( stream );
    wxInt32 n = 0;
    textStream &gt;&gt; n;
    for (int i = 0; i &lt; n; i++)
    {
        DoodleLine *line = new DoodleLine;
        textStream
            &gt;&gt; line->m_x1
            &gt;&gt; line->m_y1
            &gt;&gt; line->m_x2
            &gt;&gt; line->m_y2;
        GetLines().Append(line);
    }
    return stream;
}
void DoodleSegment::Draw(wxDC *dc)
{
    wxList::compatibility_iterator node = GetLines().GetFirst();
    while (node)
    {
        DoodleLine *line = (DoodleLine *)node->GetData();
        dc->DrawLine(line->m_x1, line->m_y1, line->m_x2, line->m_y2);
        node = node->GetNext();
    }
```

```
}
```

到目前为止,我们还没有介绍怎样将doodle片断增加到我们的文档中,除了从文件读取以外.我们需要将那些用来响应鼠标和键盘操作,以更改文档内容的命令代码模型化,这是实现重做/撤消操作的关键.DoodleCommand是一个继承自wxCommand的类,它实现了虚函数Do和Undo,这些函数将被框架在合适的时候调用.因此,我们将不会直接更改文档内容,取而代之的是在相应的事件处理函数中,创建一个一个的DoodleCommand对象,并将这些对象提交给文档命令处理器(一个wxCommandProcessor类的实例)处理.文档命令处理器在执行这些命令前会自动将这些命令保存在一个重做/撤消堆栈中.文档命令处理器对象是在文档被初始化的时候被框架自动创建的,因此在这个例子中你看不到显式创建这个对象的代码.

下面是DoodleCommand类的声明:

```
/*
 * 一个doodle命令
 */
class DoodleCommand: public wxCommand
{
public:
    DoodleCommand(const wxString& name, int cmd, DoodleDocument *doc, DoodleSegment *seg)
    ~DoodleCommand();
    /// Overrides
    virtual bool Do();
    virtual bool Undo();
    /// 重做和撤消的命令是对称的,因此将它们组合在一起.
    bool DoOrUndo(int cmd);
protected:
    DoodleSegment*  m_segment;
    DoodleDocument* m_doc;
    int             m_cmd;
};
/*
 * Doodle命令标识符
 */
#define DOODLE_CUT          1
#define DOODLE_ADD          2
```

我们定义了两种类型的命令: DOODLE_ADD和DOODLE_CUT.用户可以删除最后一次的绘画操作或者增加新的绘画操作.这里我们的两个命令都使用同一个类,不过这不是必须的.每一个命令对象都会保存一个文档指针,一个DoodleSegment(代表一次绘画操作)指针和一个命令标识符.下面是DoodleCommand 类的实现部分:

```
/*
 * DoodleCommand
 */
DoodleCommand::DoodleCommand(const wxString& name, int command,
                              DoodleDocument *doc, DoodleSegment *seg):
    wxCommand(true, name)
{
    m_doc = doc;
    m_segment = seg;
    m_cmd = command;
}
DoodleCommand::~DoodleCommand()
{
```

```
    if (m_segment)
        delete m_segment;
}
bool DoodleCommand::Do()
{
    return DoOrUndo(m_cmd);
}
bool DoodleCommand::Undo()
{
    switch (m_cmd)
    {
    case DOODLE_ADD:
        {
            return DoOrUndo(DOODLE_CUT);
        }
    case DOODLE_CUT:
        {
            return DoOrUndo(DOODLE_ADD);
        }
    }
    return true;
}
bool DoodleCommand::DoOrUndo(int cmd)
{
    switch (cmd)
    {
    case DOODLE_ADD:
        {
            wxASSERT( m_segment != NULL );

            if (m_segment)
                m_doc->GetDoodleSegments().Append(m_segment);
            m_segment = NULL;
            m_doc->Modify(true);
            m_doc->UpdateAllViews();
            break;
        }
    case DOODLE_CUT:
        {
            wxASSERT( m_segment == NULL );

            // Cut the last segment
            if (m_doc->GetDoodleSegments().GetCount() &gt; 0)
            {
                wxList::compatibility_iterator node = m_doc->GetDoodleSegments().GetLast(

                m_segment = (DoodleSegment *)node->GetData();
                m_doc->GetDoodleSegments().Erase(node);

                m_doc->Modify(true);
                m_doc->UpdateAllViews();
            }
            break;
        }
    }
    return true;
}
```

因为在我们的例子中Do和Undo操作使用共用的代码,我们直接使用一个DoOrUndo函数来实现所有的操作.如果我们被要求执行 DOODLE_ADD的撤消操作,我们可以直接执行 DOODLE_CUT,而要执行DOODLE_CUT的撤消操作,我们则直接执行 DOODLE_ADD.

当增加一个绘画片断(或者对某个Cut命令执行撤消操作)时,DoOrUndo函数所做的事情就是把这个绘画片断增加到文档的绘画片断列表,并且将自己内部的绘画片断的指针清除,以便在释放这个命令对象的时候不需要释放这个绘画片断对象.相应的,当执行Cut操作(或者Add的Undo 操

作)的时候,将文档的片断列表中的最后一个片断从列表中移除,并且保存其指针,以便用于相应的恢复操作.DoOrUndo函数做的另外一件事情是将文档标记为已修改状态(以便在应用程序退出时提醒用户保存文档)以及告诉文档需要更新和自己相关的所有的视图.

要定义自己的视图类,你需要实现wxView的派生类,同样的,需要使用动态创建的宏,并至少重载OnCreate,OnDraw,OnUpdate和OnClose函数.

OnCreate函数在视图和文档对象刚被创建的时候调用,你应该执行的动作包括:创建frame窗口,使用SetFrame函数将其和当前视图绑定.

OnDraw函数的参数为一个wxDC指针,用来实现窗口绘制操作. 实际上,这一步不是必须的,但是一旦你不使用重载OnDraw函数的方法来实现窗口绘制,默认的打印/预览机制将不能正常工作.

OnUpdate函数的参数是一个指向导致这次更新操作的视图的指针以及一个指向一个用来帮助优化视图更新操作的对象的指针.这个函数在视图需要被更新的时候调用,这通常意味着由于执行某个文档命令导致相关视图需要更新,或者应用程序显式调用了wxDocument::UpdateAllViews函数.

OnClose函数在视图需要被关闭的时候调用,默认的实现是调用wxDocument::OnClose函数关闭视图绑定的文档.

下面是DoodleView的类声明.我们重载了前面介绍的四个函数,并且增加了一个DOODLE_CUT命令的处理函数.为什么这里没有DOODLE_ADD命令的处理函数,是因为绘画片断是随着鼠标的操作而增加的,因此DOODLE_ADD命令对应的视图动作已经在 DoodleCanvas对象的鼠标处理函数中实现了.我们很快就会看到.

```
/*
 * DoodleView是文档和窗口之间的桥梁.
 */
class DoodleView: public wxView
{
    DECLARE_DYNAMIC_CLASS(DoodleView)
    DECLARE_EVENT_TABLE()
public:
    DoodleView() { m_frame = NULL; }
    ~DoodleView() {};
    /// 当文档被创建的时候调用
    virtual bool OnCreate(wxDocument *doc, long flags);
    /// 当需要绘制文档的时候被调用
    virtual void OnDraw(wxDC *dc);
    /// 当文档需要更新的时候被调用
    virtual void OnUpdate(wxView *sender, wxObject *hint = NULL);
    /// 当视图被关闭的时候调用
    virtual bool OnClose(bool deleteWindow = true);
    /// 用于处理Cut命令
    void OnCut(wxCommandEvent& event);
private:
    DoodleFrame*    m_frame;
};
```

下面的代码是其实现部分:

```
IMPLEMENT_DYNAMIC_CLASS(DoodleView, wxView)
BEGIN_EVENT_TABLE(DoodleView, wxView)
    EVT_MENU(DOODLE_CUT, DoodleView::OnCut)
END_EVENT_TABLE()
// 当视图被创建的时候需要做的动作
bool DoodleView::OnCreate(wxDocument *doc, long WXUNUSED(flags))
{
    // 将当前主窗口和视图绑定
    m_frame = GetMainFrame();
    SetFrame(m_frame);
    m_frame->GetCanvas()->SetView(this);
    // 让视图管理器感知当前视图
    Activate(true);
    // 初始化编辑菜单中的重做/撤消项目
    doc->GetCommandProcessor()->SetEditMenu(m_frame->GetEditMenu());
    doc->GetCommandProcessor()->Initialize();
    return true;
}
// 这个函数被默认的打印/打印预览以及窗口绘制函数共用
void DoodleView::OnDraw(wxDC *dc)
{
    dc->SetFont(*wxNORMAL_FONT);
    dc->SetPen(*wxBLACK_PEN);
    wxList::compatibility_iterator node = ((DoodleDocument *)GetDocument
())->GetDoodleSegments().GetFirst();
    while (node)
    {
        DoodleSegment *seg = (DoodleSegment *)node->GetData();
        seg->Draw(dc);
        node = node->GetNext();
    }
}
void DoodleView::OnUpdate(wxView *WXUNUSED(sender), wxObject *WXUNUSED(hint))
{
    if (m_frame && m_frame->GetCanvas())
        m_frame->GetCanvas()->Refresh();
}
// 清除用于显式这个视图的窗口
bool DoodleView::OnClose(bool WXUNUSED(deleteWindow))
{
    if (!GetDocument()->Close())
        return false;
    // 清除画布
    m_frame->GetCanvas()->ClearBackground();
    m_frame->GetCanvas()->SetView(NULL);
    if (m_frame)
        m_frame->SetTitle(wxTheApp->GetAppName());
    SetFrame(NULL);
    // 告诉文档管理器不要再给我发送任何事件了.
    Activate(false);
    return true;
}
void DoodleView::OnCut(wxCommandEvent& WXUNUSED(event))
{
    DoodleDocument *doc = (DoodleDocument *)GetDocument();
    doc->GetCommandProcessor()->Submit(
        new DoodleCommand(wxT("Cut Last Segment"), DOODLE_CUT, doc, NULL));
}
```

第4步: 定义你的窗口类

通常你需要创建特定的编辑窗口来维护你视图中的数据.在我们的例子中,DoodleCanvas用来显示对应的数据,和相关的事件交互等,wxWidgets的事件处理机制也要求我们最好创建一个新的派生类.DoodleCanvas类的声明如下:

```
/*
 * DoodleCanvas是用来显示文档的窗口类
 */
class DoodleView;
class DoodleCanvas: public wxScrolledWindow
{
    DECLARE_EVENT_TABLE()
public:
    DoodleCanvas(wxWindow *parent, const wxPoint& pos,
                 const wxSize& size, const long style);
    /// 绘制文档内容
    virtual void OnDraw(wxDC& dc);
    /// 处理鼠标事件
    void OnMouseEvent(wxMouseEvent& event);
    /// 设置和获取视图对象
    void SetView(DoodleView* view) { m_view = view; }
    DoodleView* GetView() const { return m_view; }
protected:
    DoodleView *m_view;
};
```

DoodleCanvas包含一个指向对应视图对象的指针(通过DoodleView::OnCreate函数初始化),以便在绘画和鼠标事件处理函数中使用.下面是这个类的实现部分:

```
/*
 * Doodle画布的实现
 */
BEGIN_EVENT_TABLE(DoodleCanvas, wxScrolledWindow)
    EVT_MOUSE_EVENTS(DoodleCanvas::OnMouseEvent)
END_EVENT_TABLE()
// 构造函数部分
DoodleCanvas::DoodleCanvas(wxWindow *parent, const wxPoint& pos,
                           const wxSize& size, const long style):
    wxScrolledWindow(parent, wxID_ANY, pos, size, style)
{
    m_view = NULL;
}
// 定制重绘行为
void DoodleCanvas::OnDraw(wxDC& dc)
{
    if (m_view)
        m_view->OnDraw(& dc);
}
// 这个函数实现了主要的涂鸦操作,主要用了鼠标左键事件.
void DoodleCanvas::OnMouseEvent(wxMouseEvent& event)
{
    // 上一次的位置
    static int xpos = -1;
    static int ypos = -1;
    static DoodleSegment *currentSegment = NULL;
    if (!m_view)
        return;
    wxClientDC dc(this);
    DoPrepareDC(dc);
    dc.SetPen(*wxBLACK_PEN);
    // 将滚动位置计算在内
    wxPoint pt(event.GetLogicalPosition(dc));
    if (currentSegment && event.LeftUp())
    {
        if (currentSegment->GetLines().GetCount() == 0)
        {
            delete currentSegment;
            currentSegment = NULL;
        }
        else
        {
            // 当鼠标左键释放的时候我们获得一个绘画片断,因此需要增加这个片断
            DoodleDocument *doc = (DoodleDocument *) GetView()->GetDocument();
            doc->GetCommandProcessor()->Submit(
                new DoodleCommand(wxT("Add Segment"), DOODLE_ADD, doc, currentSegment));
            GetView()->GetDocument()->Modify(true);
            currentSegment = NULL;
        }
    }
    if (xpos &gt; -1 && ypos &gt; -1 && event.Dragging())
    {
        if (!currentSegment)
            currentSegment = new DoodleSegment;
        DoodleLine *newLine = new DoodleLine;
        newLine->m_x1 = xpos;
        newLine->m_y1 = ypos;
        newLine->m_x2 = pt.x;
        newLine->m_y2 = pt.y;
        currentSegment->GetLines().Append(newLine);
        dc.DrawLine(xpos, ypos, pt.x, pt.y);
    }
    xpos = pt.x;
    ypos = pt.y;
}
```

正如你看到的那样,当鼠标处理函数检测到一个新的绘画片断被创建的时候,它提交给对应的文档对象一个DOODLE_ADD命令,这个命令将被保存以便支持撤消(以及将来的重做)动作.在我们的例子中,它被保存在文档的绘画片断列表中.

第5步,使用wxDocManager和wxDocTemplate

你需要在应用程序的整个生命周期内维持一个wxDocManager实例,这个实例负责整个文档视图框架的协调工作.

你也需要至少一个wxDocTemplate对象.这个对象用来实现文档视图模型中文档和视图相关联的那部分工作.每一个文档/视图对, 对应一个wxDocTemplate对象,wxDocManager对象将管理一个wxDocTemplate对象的列表以便用来创建文档和视图. wxDocTemplate对象知道怎样的文件扩展名对应目前的文档对象以及怎样创建相应的文档或者视图对象等.

举例来说,如果我们的Doodle文档支持两种视图:图形视图和绘图片断列表视图,那么我们就需要创建两种视图对象 (DoodleGraphicView和DoodleListView),相应的我们也需要创建两种文档模板对象,一个用于图形视图,一个用于列表视图. 你可以给这两个wxDocTemplate使用同样的文档类和同样的文件扩展名,但是传递不同的视图类型.当用户点击应用程序的打开菜单时,文件选择对话框将额外显示一组可用的文件过滤器,每一个过滤器对应一个wxDocTemplate类型,当某个文件被选中打开的时候,wxDocManager将使用对应的wxDocTemplate创建相应的文档类和视图类.同样的逻辑也被应用于创建新对象的时候.当然,在我们的例子中,只有一种wxDocManager对象,因此打开和新建文档的对话框就显得简单一些了.

你可以在你的应用程序种存储一个wxDocManager指针,但是对于wxDocTemplate通常没有这个必要,因为后者是被wxDocManager管理和维护的.下面是我们的DoodleApp类的定义部分:

```
/*
 *声明一个应用程序类
 */
class DoodleApp: public wxApp
{
public:
    DoodleApp();
    virtual bool OnInit();
    virtual int OnExit();
private:
    wxDocManager* m_docManager;
};
DECLARE_APP(DoodleApp)
```

在DoodleApp的实现部分,我们在OnInit函数种创建wxDocManager对象和一个和我们的DoodleDocument和DoodleView绑定的wxDocTemplate对象.我们给wxDocTemplate传递的参数包括: wxDocManager对象,描述字符串,文件过滤器(在文件对话框种使用),默认打开目录(在文件对话框种使用),默认的文件扩展名(.drw,用来区分我们的文件类型)以及我们的文档和视图类型以及对应的类型信息.DoodleApp的实现部分如下所示:

```
IMPLEMENT_APP(DoodleApp)
DoodleApp::DoodleApp()
{
    m_docManager = NULL;
}
bool DoodleApp::OnInit()
{
    // 创建一个wxDocManager
    m_docManager = new wxDocManager;
    // 创建我们需要的wxDocTemplate
    (void) new wxDocTemplate(m_docManager, wxT("Doodle"), wxT("*.drw"), wxT(""), wxT
("drw"), wxT("Doodle Doc"), wxT("Doodle View"),
        CLASSINFO(DoodleDocument), CLASSINFO(DoodleView));
    // 在Mac系统上登记文档类型
#ifdef __WXMAC__
    wxFileName::MacRegisterDefaultTypeAndCreator( wxT("drw") , 'WXMB' , 'WXMA' ) ;
#endif
    // 对于我们的单文档界面,我们只支持最多同时打开一个文档
    m_docManager->SetMaxDocsOpen(1);
    // 创建主窗口
    DoodleFrame* frame = new DoodleFrame(m_docManager, NULL, wxID_ANY, wxT("Doodle
 Sample"), wxPoint(0, 0), wxSize(500, 400), wxDEFAULT_FRAME_STYLE);
    frame->Centre(wxBOTH);
    frame->Show(true);
    SetTopWindow(frame);
    return true;
}
int DoodleApp::OnExit()
{
    delete m_docManager;
    return 0;
}
```

因为我们只支持同时显示一个文档,我们需要通过函数SetMaxDocsOpen告诉文档管理器这一点.为了在Mac OS上提供一些额外的系统特性,我们也通过MacRegisterDefaultTypeAndCreator函数在Mac系统中注册了我们的文件类型. 这个函数的参数为文件扩展名,文档类型标识符以及创建标识符(按照惯例通常采用四个字节的字符串来标识,你也可以在苹果公司的网站上注册这种类型以避免可能存在的冲突).

Doodle例子完整的代码请参考附带光盘的examples/chap19/doodle目录.

19.2 文档/视图框架的其它能力

上一节中我们通过一个简单的例子演示了使用文档视图框架所必须的一些步骤,这一节我们来讨论这个框架中一些更深入的话题.

标准标识符

文档/视图系统支持很多默认的标识符,比如wxID_OPEN, wxID_CLOSE, wxID_CLOSE_ALL, wxID_REVERT, wxID_NEW, wxID_SAVE, wxID_SAVEAS, wxID_UNDO, wxID_REDO, wxID_PRINT和wxID_PREVIEW,为了更大的发挥框架的威力,你应该尽可能在你的菜单或者工具栏中使用这些标准的标识符.这些标识符的处理函数大多已经在wxDocManager类中实现,比如OnFileOpen,OnFileClose和OnUndo等.对应的处理函数将自动调用当前文档相应的处理函数.如果你愿意,你当然可以在你的frame窗口类或者wxDocManager的派生类中重载这些处理函数,不过通常都没有这个必要.

打印和打印预览

默认情况下,wxID_PRINT和wxID_PREVIEW使用标准的wxDocPrintout类来实现打印和打印预览,以便直接重用 wxView:: OnDraw函数.然而,这种用法的一个最大的缺陷是仅适用于只有一页的文档的情形.因此你可以创建你自己的wxPrintout类来重载标准的 wxID_PRINT和wxID_PREVIEW处理,最快速的方法的方法当然是使用wxHtmlEasyPrinting类,我们在第12章,"高级窗口类"的"HTML打印"小结有比较详细的介绍.

文件访问历史

当你的应用程序初始化的时候,可以直接通过wxConfig对象使用wxDocManager::FileHistoryLoad函数在文件菜单的最下方加载一个文件访问历史列表,也可以在应用程序退出之前使用wxDocManager::FileHistorySave函数保存这个列表.比如,要加载文件访问历史,你可以这样做:

```
// 加载文件访问历史
wxConfig config(wxT("MyApp"), wxT("MyCompany"));
config.SetPath(wxT("FileHistory"));
m_docManager->FileHistoryLoad(config);
config.SetPath(wxT("/"));
```

如果你是在创建主窗口或者其主菜单之前加载的文件访问历史,你可以显式的通过wxDocManager::FileHistoryAddFilesToMenu函数将其增加在菜单中.

你也可以通过wxFileHistory类或者你自己的方法来实现文件访问历史功能,比如有时候你可能需要为每个文档窗口实现不同的文件访问历史.

显式创建文档类

有时候你需要显式的创建一个文档对象,比如说,有时候你想打开上次显式的文档,你可以通过下面的方式直接打开一个已经存在的文档:

```
wxDocument* doc = m_docManager->CreateDocument(filename,
                                               wxDOC_SILENT);
```

或者象下面这样创建一个新的文档:

```
wxDocument* doc = m_docManager->CreateDocument(wxEmptyString,
                                               wxDOC_NEW);
```

无论是上面哪种情况,都将自动创建一个相应的视图对象.

19.3 实现Undo/Redo的策略

你的应用程序的Undo/Redo机制的实现方法通常和你文档的数据类型以及用户操作数据的方式有关.在我们的例子中,我们每次只操作一整块的数据,操作也是很简单的.然而在很多应用程序中,用户可以对多种类型的数据进行操作.在这种情况下,我们可能需要使用另外一种命令表示方法,可以称之为 CommandState(状态命令),其中包含了文档中特定对象的信息.你的命令类应该维护一个状态列表,并且也可以在其构造函数中接受这样的状态列表参数.Do和Undo操作将根据列表中的状态并将当前的命令应用到对应的状态.

实现Redo/Undo操作的关键,不外乎以每次一步的方式向前或者向后遍历每个历史命令.因此,你的Undo/Redo实现可以任意对文档状态进行快照操作,以用于将来的恢复操作.通过这种方法无论用户进行多少次向前或者向后的历史命令都没有问题,你的代码所需要作的只是怎样确定"完成"和"未完成"状态.

一个常用的方法是,在每个命令状态中保存文档中每个对象的一个拷贝,以及一个指向实际对象的指针.而Do和Undo操作只是简单的交换当前状态和历史状态.举例来说,比如对象是一个图形,用户将它的颜色从红色改为蓝色.应用程序于是创建了一个新的标识符用来指示当前红色的对象,但是它将其内部的颜色属性由红色设置为蓝色.当这个命令第一次执行的时候,Do函数制作一个当前的红色对象的拷贝,并且对其应用这个新的命令(包括那个蓝颜色), 并将其置为可视状态,然后重绘这个对象,Undo作的也是同样的事情,它制作一个当前蓝色对象的拷贝,然后对齐应用原来的状态(红色),然后重绘这个对象.因此Do和Undo函数其实是完全一样的.不光如此,这个函数还可以用于除更改颜色以外的其它操作,因为整个对象包括其属性在内都被制作了一份拷贝, 你可以通过给你的应用程序所能编辑的对象单独实现各自实现赋值和拷贝操作的构造函数的方法,使得这整个过程更直观.

让我们举个例子,以便说的更明白些.假如说,我们的文档包含多种形体,用户可以更改当前选中的所有形体的颜色.我们可以在相应的更改颜色菜单命令处理函数(比如ShapeView::OnChangeColor)中,在这个命令应用于整个文档之前,为当前选种的各个对象创建一个新的状态拷贝,如下所示:

```
// 更改当前选中的形体的颜色
void ShapeView::OnChangeColor(wxCommandEvent& event)
{
    wxColour col = GetSelectedColor();
    ShapeCommand* cmd = new ShapeCommand(wxT("Change color"));
    ShapeArray arrShape;
    for (size_t i = 0; i &lt; GetSelectedShapes().GetCount(); i++)
    {
        Shape* oldShape = GetSelectedShapes()[i];
        Shape* newShape = new Shape(*oldShape);
        newShape->SetColor(col);
        ShapeState* state = new ShapeState(SHAPE_COLOR, newShape, oldShape);
        cmd->AddState(state);
    }
    GetDocument()->GetCommandProcessor()->SubmitCommand(cmd);
}
```

因为对于我们的ShapeState对象来说,Do和Undo的操作都是一样的,因此我们可以使用一个共用的DoAndUndo函数,用来进行状态交换的动作:

```
// 不完整的DoAndUndo实现
// 对于某些命令来说,Do和Undo共用同样的代码
void ShapeState::DoAndUndo(bool undo)
{
    switch (m_cmd)
    {
    case SHAPE_COLOR:
    case SHAPE_TEXT:
    case SHAPE_SIZE:
        {
            Shape* tmp = new Shape(m_actualShape);
            (* m_actualShape) = (* m_storedShape);
            (* m_storedShape) = (* tmp);
            delete tmp;
            // 进行重绘动作
            ...
            break;
        }
    }
}
```

上面的代码中我们没有列出ShapeCommand::Do和ShapeCommand:Undo函数,这两个函数所做的事情就是遍历其成员列表中的所有状态的相应函数.

(译者注:这一小节翻译的似曾相识,云里雾里)

第十九章小结

在这一章里,我们看到了怎样通过wxWidgets提供的文档/视图框架来简化这种类型的应用程序的编程,让wxWidgets来自动处理那些显示文件对话框或者创建文档和视图对象这种琐碎的工作.你也对如果实现Undo/Redo机制有了一个大概的印象.这个机制你应该在你的应用程序的初期就给予充分的考虑,否则到了应用程序的开发后期,这些功能的实现可能导致你重写大部分的代码.

在我们的最后一章中,我们将讨论一下如何让你的应用程序显得更完美.

第二十章完善你的应用程序

一个简朴可用的应用程序和一个方便优雅的应用程序之间有很大的不同.如果只是内部使用,基本的不带有很多修饰的应用程序也许是足够的,但是如果你的应用程序打算分发给世界范围的很多人使用,你应该让它作到更令人信服和更容易使用,就像大多数商业软件公司制作的软件一样.你的软件应该遵守很多约定俗称的惯例和标准,比如提供配置对话框和联机帮助等.在这本书的最后一章,我们将讨论下面几个话题,它们可以让你的软件显得更专业:

- 单实例程序还是多实例程序? 怎样阻止同时运行你的程序的多个实例.
- 更改事件处理机制.怎样更改事件的执行顺序.
- 降低闪烁.怎样通过降低闪烁的方法提高你的应用程序可视界面的观感.
- 实现在线帮助.将给你提供一些实现各种联机帮助的建议.
- 解析命令行参数. 让你的用户可以通过命令行参数更直接的控制你的应用程序的行为.
- 保存应用程序所用的资源. 介绍你打包各种资源文件的方法.
- 调用别的应用程序. 从简单的调用别的程序执行,到控制别的应用程序的输入和输出方法.
- 管理应用程序设置. 通过wxConfig类来保存和加载应用程序设置以及和应用程序设置相关的一些提示.
- 应用程序安装.一些关于怎么让你的用户可以方便的在它们的平台上安装你的软件包的建议.
- 遵循用户界面设计规范. 关于各个平台用户界面设计规范方面的一些建议.

20.1 单个实例和多个实例

依照你的应用程序的性质的不同,你可能允许你的用户同时运行多个你的应用程序的实例,或者你也可能希望同时只能存在一个你的应用程序的实例,如果用户试图打开第二个或者试图通过资源管理器打开多个和你的应用程序关联的文档,它们都将只看到一个应用程序在运行.一种很通用的作法是,让你的用户根据他的工作习惯,自己选择是否允许运行应用程序的多个实例.不过同时运行多个实例有一个大问题是,哪个程序实例首先将配置文件写入磁盘是不一定的,因此用户可能会丢失它们的设置.而且多个实例运行对一些新手来说也有麻烦,它们可能并没有意识到它们已经运行了应用程序的多个实例.允许同时运行应用程序的多个实例在所有的平台上都是默认选项(除了在Mac平台上通过Finder(查找器)来启动应用程序打开文档的时候),因此,如果你不希望你的应用程序可以运行多个实例, 你需要一些额外的代码.

在Mac OS(也仅在这个系统)上,使用单一实例打开多个文档是非常容易的.你只需要重载MacOpenFile函数,这个函数采用一个wxString类型的文件名作为参数,这个函数将在Mac Os的Finder打开和这个应用程序关联的文档的时候被调用.如果当前还没有运行任何这个应用程序的实例,Mac系统会先启动一个实例,然后再调用这个函数(这个大多数系统不同,在别的系统上,打开文档通常是通过调用应用程序并且将文件名作为参数的方法).如果你使用的是文档/试图框架,你可能并不需要重载这个函数,因为其在Mac OsX上的实现代码如下:

```
void wxApp::MacOpenFile(const wxString& fileName)
{
    wxDocManager* dm = wxDocManager::GetDocumentManager() ;
    if ( dm )
        dm->CreateDocument(fileName, wxDOC_SILENT) ;
}
```

然后,即使在Mac Os上,用户还是可以通过直接多次运行程序的方法运行你的应用程序的多个实例. 如果你想检测和禁止运行超过一个应用程序实例,你可以在程序运行之初使用wxSingleInstanceChecker类.这个对象将保持在应用程序的整个生命周期,因此,在你的OnInit函数中,调用其IsAnotherRunning函数检测是否已经有别的实例正在运行,如果返回true,你可以在警告你的用户之后,立刻退出应用程序.如下所示:

```
bool MyApp::OnInit()
{
    const wxString name = wxString::Format(wxT("MyApp-%s"),
                                   wxGetUserId().c_str());
    m_checker = new wxSingleInstanceChecker(name);
    if ( m_checker->IsAnotherRunning() )
    {
        wxLogError(_("Program already running, aborting."));
        return false;
    }
    ... more initializations ...
    return true;
}
int MyApp::OnExit()
{
    delete m_checker;
    return 0;
}
```

但是,如果你想把旧的实例带到前台,或者你想使用旧的实例打开传递给你的新的实例作为命令行参数的文件,该怎么办呢?一般说来,这需要在这两个实例间进行通讯.我们可以使用wxWidgets提供的高层进程间通讯类来实现.

在下面的例子中,我们将实现应用程序多个实例之间的通讯,以便允许第二个实例询问第一个实例是否它自己打开相应的文件,还是将自己提到前台以便提醒用户它已经请求用这个应用程序来打开文件,下面的代码声明了一个连接类,这个类将被两个实例使用.一个服务器类将被老的实例使用以便监听别的实例的连接请求,一个客户端类将被后来的实例使用以便和老的实例通讯.

```
#include "wx/ipc.h"
// Server类,用来监听连接请求
class stServer: public wxServer
{
public:
    wxConnectionBase *OnAcceptConnection(const wxString& topic);
};
// Client类,在OnInit函数中被后来的实例使用
class stClient: public wxClient
{
public:
    stClient() {};
    wxConnectionBase *OnMakeConnection() { return new stConnection; }
};
// Connection类,被两个实例同时使用以实现通讯
class stConnection : public wxConnection
{
public:
    stConnection() {}
    ~stConnection() {}
    bool OnExecute(const wxString& topic, wxChar*data, int size,
                   wxIPCFormat format);
};
```

OnAcceptConnection函数在老实例(Server)中当有新实例(Client)进行连接请求的时候被调用.我们应该首先检查老实例中没有显示任何模式对话框,因为如果有模式对话框,就不可能有别的行为可以引起用户的注意.

```
// 接收到了来自别的实例的连接请求
wxConnectionBase *stServer::OnAcceptConnection(const wxString& topic)
{
    if (topic.Lower() == wxT("myapp"))
    {
        // 检查没有活动的模式对话框
        wxWindowList::Node* node = wxTopLevelWindows.GetFirst();
        while (node)
        {
            wxDialog* dialog = wxDynamicCast(node->GetData(), wxDialog);
            if (dialog && dialog->IsModal())
            {
                return false;
            }
            node = node->GetNext();
        }
        return new stConnection();
    }
    else
        return NULL;
}
```

OnExecute函数在客户端实例对其连接对象调用Execute函数的时候被调用. OnExecute函数可以有一个空的参数,这表示它需要将自己提到前台就可以了,否则,它需要检测参数中的文件名指示的文件是否已经被它打开,如果已经打开,将这个文件显示给用户,否则,就打开这个文件,再将其显示给用户.

```
// 打开别的实例传来的文件参数.
bool stConnection::OnExecute(const wxString& WXUNUSED(topic),
                             wxChar *data,
                             int WXUNUSED(size),
                             wxIPCFormat WXUNUSED(format))
{
    stMainFrame* frame = wxDynamicCast(wxGetApp().GetTopWindow(), stMainFrame);
    wxString filename(data);
    if (filename.IsEmpty())
    {
        // 只需要提升主窗口
        if (frame)
            frame->Raise();
    }
    else
    {
        // 检查文件是否已经打开并且将其显示给用户
        wxNode* node = wxGetApp().GetDocManager()->GetDocuments().GetFirst();
        while (node)
        {
            MyDocument* doc = wxDynamicCast(node->GetData(),MyDocument);
            if (doc && doc->GetFilename() == filename)
            {
                if (doc->GetFrame())
                    doc->GetFrame()->Raise();
                return true;
            }
            node = node->GetNext();
        }
        wxGetApp().GetDocManager()->CreateDocument(
                                 filename, wxDOC_SILENT);
    }
    return true;
}
```

在OnInit函数中,应用程序应该首先象前面介绍的那样使用wxSingleInstanceChecker检查是否已经运行了多个实例,如果没有别的实例运行,这个实例可以将自己设置位一个Server,等待别的应用程序实例的连接请求,如果已经有实例在运行,就创建一个和那个实例的连接,第二个实例请求第一个实例打开自己被请求的文件或者提升其主窗口.下面是相关的代码:

```
bool MyApp::OnInit()
{
    wxString cmdFilename; // code to initialize this omitted
    ...
    m_singleInstanceChecker = new wxSingleInstanceChecker(wxT("MyApp"));
    // 如果使用单实例,用IPC检测是否有别的实例.
    if (!m_singleInstanceChecker->IsAnotherRunning())
    {
        // 创建一个服务器
        m_server = new stServer;
        if ( !m_server->Create(wxT("myapp") )
        {
            wxLogDebug(wxT("Failed to create an IPC service."));
        }
    }
    else
    {
        wxLogNull logNull;
        // OK, 已经有一个实例了,创建一个和它之间的连接,然后在自己退出之前发送文件名
        stClient* client = new stClient;
        // 下面的参数在使用DDE的时候被忽略,在使用基于TCP/IP的类的时候代表主机名.
        wxString hostName = wxT("localhost");
        // 创建连接
        wxConnectionBase* connection =
                    client->MakeConnection(hostName,
                                wxT("myapp"), wxT("MyApp"));
        if (connection)
        {
            // 请求那个已经存在的实例打开文件或者提升它自己
            connection->Execute(cmdFilename);
            connection->Disconnect();
            delete connection;
        }
        else
        {
            wxMessageBox(wxT("Sorry, the existing instance may be
            too busy too respond.\nPlease close any open dialogs and retry."),
                wxT("My application"), wxICON_INFORMATION|wxOK);
        }
        delete client;
        return false;
    }
    ...
    return true;
}
```

如果你想要了解更多这里用到的进程间通讯的细节,你可以在wxWidgets自带的utils/helpview/src目录中,找到另外一个用在独立的wxWidgets帮助阅读器中的例子,在那个例子中,别的应用程序会通过进程间通讯的方式请求帮助阅读器程序打开某个帮助文件,另外在wxWidgets的samples/ipc例子中,也演示了wxServer, wxClient和wxConnection的用法.

20.2 更改事件处理机制

在通常情况下,wxWidgets将事件发送到产生这个事件的窗口(或者别的事件处理器).对于命令事件,还将以特定的方式遍历整个窗口继承树(详情参考附录H,"wxWidgets的事件处理机制")来处理.举例来说,如果你点击工具条上的任何一个工具按钮,产生的事件将首先发送给这个工具按钮的事件表处理,然后是包含这个工具条的frame窗口的时间表,然后是整个应用程序类的事件表.通常情况下,这样的作法是满足要求的,但是如果有时候,你希望超过一个控件都可以使用工具条上的拷贝命令的时候,就会有一些问题,比如,你的主程序中有一个主窗口(假设是一个绘画程序)和一个文本编辑框,这个文本编辑框可能永远也无法处理工具条上的拷贝命令,因为这个命令并不会调用这个编辑框的事件处理函数.在这种情况下,你可能希望事件能够首先交给当前处于活动状态的控件处理,然后再按照正常的处理顺序执行.这样,如果当前的活动控件内部实现了针对这个事件的处理函数(比如wxID_COPY),那么这个函数就将被调用,否则就在窗口继承树中向上查找对应的处理函数,直到它找到一个这样的函数.这种命令总是针对当前活动控件的作法会更符合用户的使用习惯.

我们可以通过下面的方法重载主窗口的ProcessEvent函数,以便拦截命令事件并将其首先交给当前活动的控件处理,如下所示:

```
bool MainFrame::ProcessEvent(wxEvent& event)
{
    static wxEvent* s_lastEvent = NULL;
    // 避免死循环
    if (& event == s_lastEvent)
        return false;
    if (event.IsCommandEvent() &&
       !event.IsKindOf(CLASSINFO(wxChildFocusEvent)) &&
       !event.IsKindOf(CLASSINFO(wxContextMenuEvent)))
    {
        s_lastEvent = & event;
        wxControl *focusWin = wxDynamicCast(FindFocus(), wxControl);
        bool success = false;
        if (focusWin)
            success = focusWin->GetEventHandler()
                                ->ProcessEvent(event);
        if (!success)
            success = wxFrame::ProcessEvent(event);
        s_lastEvent = NULL;
        return success;
    }
    else
    {
        return wxFrame::ProcessEvent(event);
    }
}
```

就目前的情况来看,这种作法在那些当前活动控件可能为一个wxTextCtrl控件的时候显的更有用(在大多数平台上), wxWidgets为这种控件实现了多种内置的命令,包括wxID_COPY, wxID_CUT, wxID_PASTE, wxID_UNDO和wxID_REDO,还实现了一些默认的用户界面行为.不

过,你也可以给任意的控件通过实现其派生类的方式增加这些默认的事件处理函数,比如说wxStyledTextCtrl控件(参考examples/chap20/pipedprocess中的例子,这个例子为wxStyledTextCtrl控件提供了这种增强处理).

20.3 降低闪烁

闪烁问题是所有GUI程序员心中永远的痛.通常所有的应用程序都需要想办法来降低可能的闪烁,下面我们就这方面的话题来给您一些有益的提示.

在Windows平台上,如果你的窗口正在使用wxFULL_REPAINT_ON_RESIZE类型,尽量去掉它.这将导致窗口系统只重绘那些由于改变大小或者别的操作而被"破坏"的那部分而不是整个窗口,这将降低擦除和重画操作的范围.否则,即使窗口只是改变一点点的大小,整个窗口都将被重画,从而导致屏幕闪烁.不过如果你的程序中,主窗口显式的内容是和窗口的大小有关的,这种情况下,这种方法毫无用处,因为整个窗口必须被重新绘制.

有时候你可以设置wxCLIP_CHILDREN类型,这将阻止一个窗口发生改变的时候同时刷新它的子窗口.不过这个类型在别的平台上没有影响.

当你在滚动窗口上绘制的时候,你可以采取很多步骤来提供重绘的效率以减小闪烁.首先,你需要优化你获得绘制数据的方式:你只需要搜集那些可以在当前的可视区域内显示的数据(参考wxWindow::GetViewStart函数和wxWindow:: GetClientSize函数手册),而且在你的重画函数内,你也可以只重画那些处于需要更新区域内的图形(参考wxWindow:: GetUpdateRegion).在设计数据结构的时候,你应该考虑怎样设计才能使得你可以快速访问到当前视图对应的数据,比如说,如果你的每个数据项都有相同的宽度的时候,你可以使用链表或者数据来存放这些数据,这比你挨个搜索所有的数据要快速的多.如果计算当前位置是一个很耗时的操作,你可以考虑对最近访问的页面的起始数据位置进行缓存.然后你可以直接通过缓存快速的进行上一页下一页这样的起始数据定位动作.你也可以为每块数据增加一个用来记录其高度的字段,以便不必每次都需要计算这个数据段的高度.

当你需要实现可以滚动的图片时,你可以使用wxWindow::ScrollWindow函数,来对窗口进行物理滚动,这将使得你只需要更新剩下的很小的区域,这将大大降低闪烁.(wxScrolledWindow类已经为你实现了这种机制,GetUpdateRegion函数将反应你需要更新的这个很小的区域.)

正如我们在第5章,"绘画和打印"中介绍的那样,你还可以定义你自己的背景擦除事件处理函数,并将其留空,以禁止控件自己清除自己的背景.然后你可以直接在旧的图片上更新整个图片(包括整个背景所在的范围),以便降低由于在绘图前擦除背景导致的屏幕闪烁.使用wxWindow::SetBackgroundStyle函数将背景类型设置为wxBG_STYLE_CUSTOM将阻止控件自作主张的更新背景.第5章还介绍了 wxBufferedDC和wxBufferedPaintDC,你可以结合前面提到的这些技术一起使用来降低闪烁.

另外一种情况是由于对一个窗口进行多次连续的更新导致窗口闪烁.wxWidgets提供了wxWindow::Freeze函数和 wxWindow::Thaw函数,这两个函数可以控制窗口在被更新的时候是否立即显示在屏幕上.比如说,在你需要往一个文本框中增加很多行文本的时候, 或者往一个列表框中增加很多个子项的时候,你可以使用这两个函数.当Thaw函数被调用的时候,窗口才会进行彻底刷新.在Windows系统和Mac OSX系统上,所有的wxWindow类都支持Freeze和Thaw函

数,而在GTK+平台上,wxTextCtrl也支持这两个函数.你也可以在自己的控件中实现这两个函数来避免过度的更新用户界面(第12章介绍的wxThumbnailCtrl例子实现了这两个函数,你可以参考 examples/chap12/thumbnail目录中的代码).

20.4 实现联机帮助

虽然你应该尽可能的将你的应用程序界面设计的非常直观,以便用户根本不需要使用联机帮助,但是除了那些最最简单的应用程序外,对于大多数应用程序来说,提供联机帮助都是一个非常重要的事情.你可以提供PDF版的帮助文件或者是HTML格式的帮助文件以便用户可以使用他最习惯的方式浏览,不过如果你能够借助于某种帮助制作手段,使得你的联机帮助中的主题直接和对话框或者你的主窗口中的控件联系起来,这会让你的用户感觉更好.

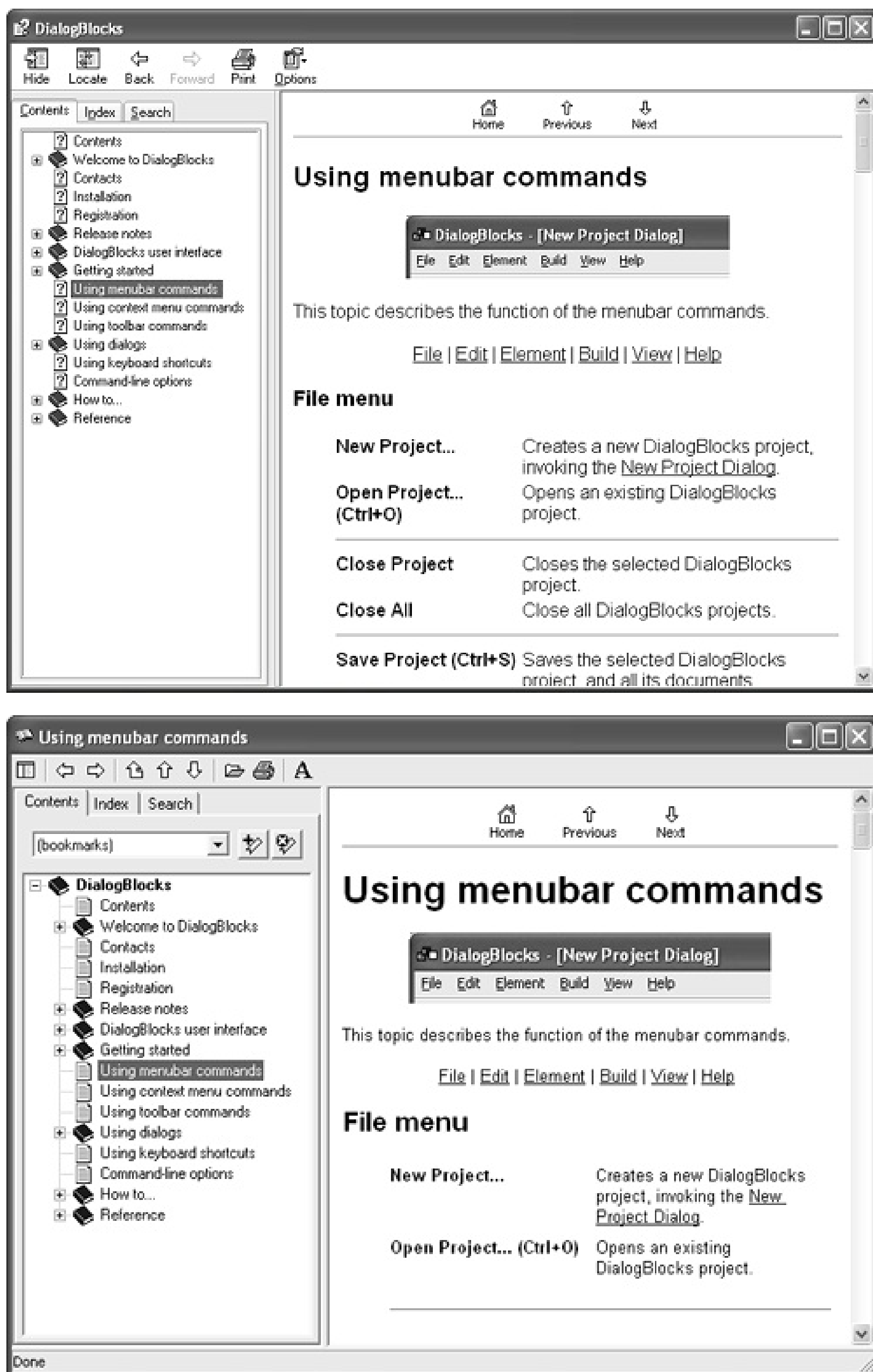
wxWidgets提供了帮助控制器,你的应用程序可以使用它来加载和显示帮助文件中的主题,它主要包括下面几个类:

- wxWinHelpController,这个类用来提供对windows平台的基于RTF格式的帮助文件(扩展名为.hlp)的支持. 这中格式现在已经不推荐使用了,新的用来代替它的类是wxCHMHelpController.
- wxCHMHelpController, 用来提供对windows平台的MS HTML帮助文件(扩展名是.chm)的支持.
- wxWinceHelpController,用来提供对WindowsCE上的帮助文件(扩展名是.htm)的支持.
- wxHtmlHelpController, 用来提供对wxWidgets自定义的HTML帮助文件(扩展名是.htb)的支持.

wxHtmlHelpController类和别的帮助控制类是不同的,别的类都只是封装了那个平台上相应的本地实现,而这个类是和 wxWidgets的帮助实现机制集成在一起的,和主程序属于同一个进程.如果你想以不同进程的方式使用wxWidgets提供的帮助文件,你可以编译 wxWidgets自带的utils/src/helpview目录下的HelpView程序.其中文件remhelp.h和remhelp.cpp实现了一个远程帮助文件控制器(wxRemoteHtmlHelpController),你可以将它和你的应用程序链接在一起,以便你的应用程序可以远程控制位于别的进程的帮助文件控制器实例.

注意到目前为止,还没有实现对Mac OSX平台上的帮助文件的支持,在这个平台上,你可以使用通用的wxWidgets HTML格式的帮助文件.

下面的两副图演示了Windows平台上同时显示MS HTML格式的帮助文件和wxWidgets的HTML帮助文件的样子.它们两个的外观很相似,都是右边显示HTML帮助的内容,左边则显示主题的继承关系以及一个用来搜索主题的文本框.稍微有一点不同的是:wxWidgets格式的帮助控制器可以同时加载多个帮助文件.



使用帮助控制器

一般说来,你需要创建一个帮助文件控制器并且在整个应用程序的生命周期维护它,通常是在应用程序类中保存一个指针,在OnInit函数中初始化,在OnExit函数中释放.之所以使用指针是因为你可以自己控制什么时候释放这个指针,某些帮助控制器是依赖于某种动态链接库类的,这个类在应用程序对象被释放以后就不存在了.在创建这个帮助控制器指针以后,使用其Initialize指定一个帮助文件.你可以不用提供文件的扩展名, wxWidgets将会提供针对当前平台的扩展名,比如:

```
#include "wx/help.h"
#include "wx/fs_zip.h"
bool MyApp::OnInit()
{
    ...
    // wxWidgets HTML需要这个
    wxFileSystem::AddHandler(new wxZipFSHandler);
    m_helpController = new wxHelpController;
    m_helpController->Initialize(helpFilePath);
    ...
    return true;
}
int MyApp::OnExit()
{
    delete m_helpController;
    ...
    return 0;
}
```

注意这里我们让wxWidgets自己决定使用哪个帮助控制器类: wxHelpController在Windows平台上定义为wxCHMHelpController,而在别的平台上则定义为 wxHtmlHelpController.你可以在windows平台上也使用wxHtmlHelpController类,不过最好是尽可能使用本地平台提供的帮助控制器.

一旦帮助控制器初始化成功,你就可以使用下面的函数显示相应的帮助了:

```
// 显示帮助内容
m_helpController->DisplayContents();
// 显示标题为"Introduction"的主题
m_helpController->DisplaySection(wxT("Introduction"));
// 显示包含在指定文件中的帮助主题.
m_helpController->DisplaySection(wxT("wx.html"));
// 显示指定ID的主题(仅适用于WinHelp和MS HTML Help)
m_helpController->DisplaySection(500);
// 查找关键字
m_helpController->KeywordSearch(wxT("Getting started"));
```

通常情况下,你将在帮助菜单的事件处理函数中调用DisplayContents,也许在帮助菜单中你还会列举出其它的重要的主题,你可以在其对应的事件处理函数中使用DisplaySection函数,如果你希望通过标题使用DisplaySection函数,那么所有的标题必须是不重复的.

可能你还打算在所有的自定义对话框上增加一个帮助按钮,以便用来显示针对这个对话框的帮助主题.然而,这里有一个需要注意的事情:不是所有的平台都支持从一个模式对话框的事件处理函数中显示帮助.当帮助文件是通过一个外部程序显示的时候(比如Windows平台的wxCHMHelpController),你可以放心的在模式对话框中调用帮助控件,但是如果帮助控制器只是在本进程中通过一个非模式对话框显示帮助文件时(比如wxHtmlHelpController),你必须小心,因为通常我们不可以在一个模式对话框中创建一个非模式的frame窗口.模式对话框不允许你切换到另外一个非模式的对话框中.在wxGTK平台上,这种行为仅对于wxHtmlHelpController来说是可以的,但是在Mac OS X平台上,这种行为是不允许的,这时你可以使用前面介绍的使用自己编译的HelpView程序等外部程序来显示帮助文件,或者使用模式对话框来显示帮助文件,后一种方法我们稍后会进一步描述.

扩展wxWidgets HTML帮助

wxWidgets的HTML帮助系统确实是很不错的,不过它有两个问题:首先,它只能在自己的frame窗口中显示帮助,因此你不可以在你的主窗口的某个TAB控件中显示帮助文件,另外一个缺陷是前面提到过的在模式对话框中使用的问题.

为了解决这两个问题而对wxWidgets HTML帮助系统进行的一个扩展将wxWidgets的帮助显示在一个自定义的窗口中,这个窗口可以是任何类型窗口的子窗口.你可以从光盘的examples/chap20/htmlctrlex目录中或者ftp: //biolpc22.york.ac.uk/pub/contrib/helpctrlex获取它的源代码.

如果你将它集成进你的应用程序,你可以将wxHtmlHelpWindowEx窗口集成进你的主窗口,然后使用wxHtmlHelpControllerEx类来对它进行和别的控制器一样的控制来显示帮助内容,下面是一个例子:

```
#include "helpwinex.h"
#include "helpctrlex.h"
bool MyApp::OnInit()
{
    ...
    m_embeddedHelpController = new wxHtmlHelpControllerEx;
    m_embeddedHelpWindow = new wxHtmlHelpWindowEx;
    m_embeddedHelpWindow->SetController(m_embeddedHelpController);
    m_embeddedHelpController->SetHelpWindow(m_embeddedHelpWindow);
    m_embeddedHelpController->UseConfig(config, wxT("EmbeddedHelp"));
    m_embeddedHelpWindow->Create(parentWindow,
        wxID_ANY, wxDefaultPosition, wxSize(200, 100),
        wxTAB_TRAVERSAL|wxNO_BORDER, wxHF_DEFAULT_STYLE);
    m_embeddedHelpController->AddBook(wxT("book1.htb"));
    m_embeddedHelpController->AddBook(wxT("book2.htb"));
    return true;
}
int MyApp::OnExit(void)
{
    if (m_embeddedHelpController)
    {
        m_embeddedHelpController->SetHelpWindow(NULL);
        delete m_embeddedHelpController;
    }
    ...
    return 0;
}
```

而为了解决模式对话框的问题,你可以使用wxModalHelp类来在模式对话框中显示某个帮助主题.当用户看完帮助以后,需要使用关闭按钮关闭这个帮助对话框,然后焦点才会重新返回上一个模式对话框中去.下面的代码就是你所需要作的全部:

```
wxModalHelp help(parentWindow, helpFile, topic);
```

在同一个程序使用两种不同的方法来显示帮助有时候是很不方便的,这时候你可以使用下面的函数来节约一些代码:

```
// 如果modalParent不为空,则使用模式帮助显示方法,否则就使用普通的方法.
void MyApp::ShowHelp(const wxString& topic, wxWindow* modalParent)
{
#if USE_MODAL_HELP
    if (modalParent)
    {
        wxString helpFile(wxGetApp().GetFullAppPath(wxT("myapp")));
        wxModalHelp help(modalParent, helpFile, topic);
    }
    else
#endif
    {
        if (topic.IsEmpty())
            m_helpController->DisplayContents();
        else
            m_helpController->DisplaySection(topic);
    }
}
```

宏USE_MODAL_HELP应该在那些使用wxHtmlHelpController控件的平台上被定义.当你在模式对话框中显示帮助的时候,将这个对话框的指针作为ShowHelp函数的第二个参数,这样,如果需要,帮助就会显示在一个模式的对话框中,而当你不是在模式对话框中显示帮助的时候,只需要传递NULL作为ShowHelp的第二个参数.

帮助文件中的声明

大多数现代的帮助文件系统都是基于HTML格式的.为了让跨平台的帮助文件制作更容易一些,wxWidgets的HTML帮助文件使用了和MS的HTML帮助文件同样的工程,内容以及关键字文件输入格式.这可以让你在制作多平台的帮助文件时,只需要考虑一套文件就可以了.下面列举出为了创建帮助文件所需要的所有的文件:

- 一套HTML文件,每个主题是一个文件.
- 一个内容文件(扩展名是hhc)以XML的格式描述了主题的继承关系.
- 一个可选的关键字文件(扩展名是hhk)用来将关键字映射到帮助主题.
- 一个工程文件(扩展名是hhp)用来描述工程中所有的其它文件以及整个工程的各个选项.

然后,你就可以将它们编译为MS HTML帮助文件格式(扩展名为chm)或者wxWidgets的HTML帮助文件格式(扩展名是htb).对于前者,你需要使用微软的HTML帮助制作软件(Microsoft's HTML Help Workshop),它既可以从命令行调用也可以从GUI界面中被调用,而对于后者来说,你只需要使用任何你喜欢的zip压缩工具将所有的这些文件打包成一个文件,将扩展名修改为.htb就可以了.

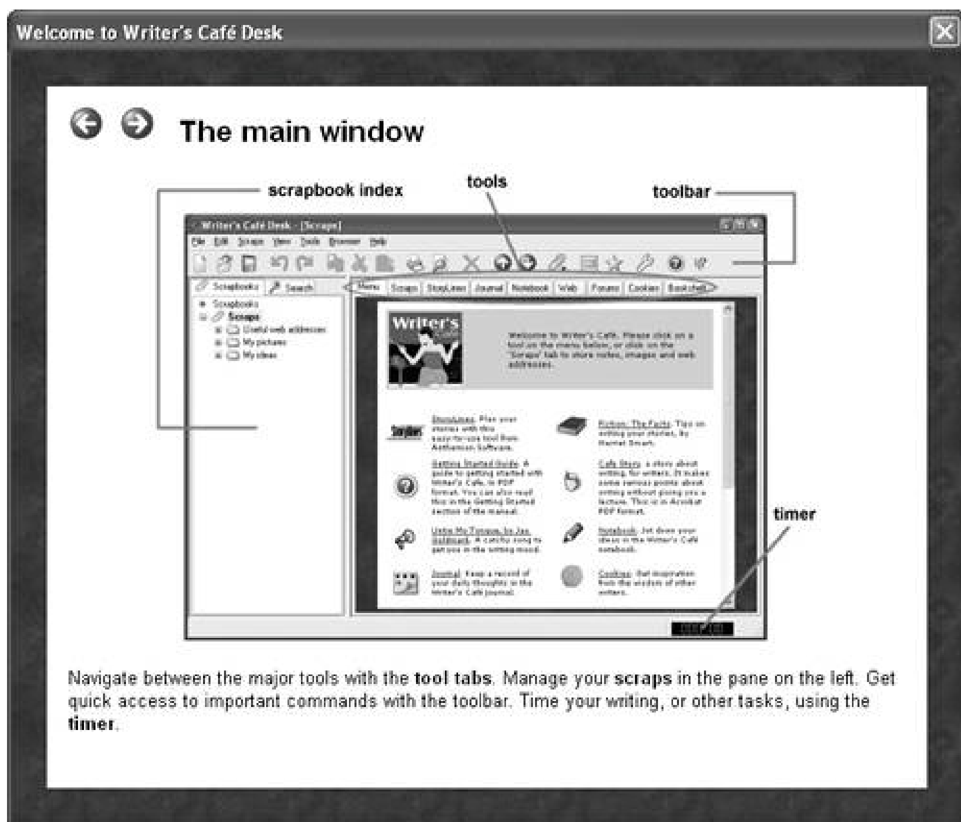
当然你可以手动制作你的帮助文件,但是使用一个工具可以节省你的时间.有很多MS HTML帮助文件的制作工具,但是它们有时候会输出不兼容于wxWidgets(不能被wxWidgets解析)的HTML标记.Anthemion Software公司的HelpBlocks软件是目前唯一的可以同时支持MS HTML格式和wxWidgets HTML格式的软件,可以用来帮助你制作帮助文件和分析关键字.

要了解好的帮助文件的结构,最好是看看别人是怎么作的.你可以考虑增加下面的这些主题:内容,欢迎,联系信息,安装信息,注册信息,发布信息,教程,菜单使用帮助,工具条使用帮助,对话框使用帮助(将所有针对对话框的主题作为子标题),快捷键,命令行参数以及故障解决等内容.

记住,应用程序中各个帮助主题的风格最好设计成各自独立的.比如教程主题,最好采用一种比较现代的风格.

其它提供帮助的手段

你也可以考虑使用别的方法给你的用户提供帮助,比如使用wxWidgets的HTML类等.Anthemion Software公司的Writer's Caf�软件使用一个模式对话框来实现一些初始选项,它是用wxHtmlWindow实现的,这些选项包括显示一个快速教程来通过一系列的HTML文件演示这个程序的用途(如下图所示).这样作的好处是不言而喻的,对于您软件的使用新手来说,这样做可以给你的用户提供一个更狭长的学习路径以便你的用户不至于一开始就被浩如烟海的帮助给淹没,从而把自己至于迷失的状态.



另外一个常用的方法是提供每日一学之类的启动提示对话框,这种方法把应用程序的功能分成一个一个的小片,然后每天学习一点点,这很符合某些人的口味.我们在第8章,"使用标准对话框"中已经介绍了怎样使用wxShowTip函数来显示这种帮助,它的参数包括一个父窗口,一个wxTipProvider指针以便告诉wxWidgets到哪里去寻找那些帮助信息,以及一个boolean变量以便指定除此显示这个对话框的时候,用于给用户选择是否显示这种帮助的复选框的初始选项.如下所示:

```
#include "wx/tipdlg.h"
...
m_tipProvider = wxCreateFileTipProvider(tipFilename, currentTip);
wxShowTip(parent, m_tipProvider, showAtStart);
```

上下文敏感帮助和工具提示

应用程序应该尽可能的提供上下文敏感帮助和工具提示.所谓工具提示是指一个很小的提示窗口,这个窗口是当鼠标在某个控件上提留一小段时间的时候探出来的.上下文敏感帮助也和它类似,不过它是由用户先点击某个帮助按钮或者是工具条上的系统按钮,然后再点击他感兴趣的控件来加以显示的.第9 章,"创建定制的对话框"中,对这些内容有比较详细的介绍,在那里我们用来介绍怎样给对话框提供帮助信息,然而这些方法不仅仅只能应用于对话框,它可以应用于任何一种窗口类型.比如用户可以在工具条或者帮助菜单中提供一个"这是什么?"的选项,以便在用户选择这个菜单或者点击这个工具按钮以后,对用户随后点击的窗口控件提供工具提示.你也不必拘泥于系统默认提供的帮助显示方法,你可以自己重载wxHelpProvider类来实现你自己的ShowHelp 函数.

一些控件支持使用更本地化的方法来提供上下文敏感帮助,如果你正在使用wxCHMHelpController控件,你可以使用它来提供上下文敏感的帮助,比如:

```
#include "wx/cshelp.h"
m_helpController = new wxCHMHelpController;
wxHelpProvider::Set(
      new wxHelpControllerHelpProvider(m_helpController));
```

wxHelpControllerHelpProvider类的实例将使用其m_helpController成员的DisplayTextPopup函数来提供上下文敏感帮助.

注意,提供上下文敏感帮助和Mac OSX的风格是格格不入的,因此在这个平台你应该忽略这种帮助.

菜单项提示

当你在菜单中加入菜单项的时候,你可以提供一个帮助信息字符串.如果这个菜单是菜单条的一部分,而这个菜单条所在的frame窗口拥有一个状态条,那么当用户鼠标在这个菜单项上划过的时候,这个帮助信息将显示在状态栏上.你可以通过wxFrame::SetStatusBarPane函数来指定显示这个帮助信息的状态条方格(如果设置为-1则将禁止显示帮助信息).这个行为是在wxFrame的默认菜单项事件 EVT_MENU_HIGHLIGHT_ALL的处理函数中实现的,因此你可以拦截这个事件来提供你自己的显示方式,比如将这个帮助信息显示在另外的一个窗口上.

20.5 解析命令行参数

允许程序在初始化的时候分析命令行参数是很有用的,对于一个文档视图架构的程序来说,你应该允许程序通过这样的方式打开文件.也可能你想让你的程序可以从命令行启动,以便进行一些自动化的工作,这时候你可以通过命令行参数告诉你的应用程序不要显示用户界面.虽然通常应用程序的大部分工作都是通过用户界面完成的,但是有时候命令行参数还是很有用的,比如用它来打开程序的调试开关.

wxWidgets提供了wxCmdLineParser类用来简化这部分的编程工作,以避免你需要直接处理wxApp::argc和 wxApp::argv.这个可以处理开关类型参数(比如-verbose),选项类型参数(比如-debug:1)以及命令参数(比如 "myfile.txt")等.对于开关类型参数和选项类型参数,它允许你设置它们的长参数形式和短参数形式,你还可以给每个参数提供一个帮助字符串,这个字符串将在需要显示使用帮助的时候打印在当前的Log目标上.

下面的例子演示了怎样使用开关类型,选项类型等各种参数:

```
#include "wx/cmdline.h"
static const wxCmdLineEntryDesc g_cmdLineDesc[] =
{
    { wxCMD_LINE_SWITCH, wxT("h"), wxT("help"),    wxT("displays help on the command line
 parameters") },
    { wxCMD_LINE_SWITCH, wxT("v"), wxT("version"), wxT("print version") },
    { wxCMD_LINE_OPTION, wxT("d"), wxT("debug"), wxT("specify a debug level") },

    { wxCMD_LINE_PARAM,  NULL, NULL, wxT("input file"), wxCMD_LINE_VAL_STRING,
 wxCMD_LINE_PARAM_OPTIONAL },
    { wxCMD_LINE_NONE }
};
bool MyApp::OnInit()
{
    // 分析命令行
    wxString cmdFilename;
    wxCmdLineParser cmdParser(g_cmdLineDesc, argc, argv);
    int res;
    {
        wxLogNull log;
        // 传递False参数以便在分析命名行发生错误的时候不显示使用帮助对话框.
        res = cmdParser.Parse(false);
    }
    // 检查是否用户正在询问使用帮助
    if (res == -1 || res &gt; 0 || cmdParser.Found(wxT("h")))
    {
        cmdParser.Usage();
        return false;
    }
    // 检查是否用户正在询问版本号
    if (cmdParser.Found(wxT("v")))
    {
#ifndef __WXMSW__
        wxLog::SetActiveTarget(new wxLogStderr);
#endif
        wxString msg;
        wxString date(wxString::FromAscii(__DATE__));
        msg.Printf(wxT("Anthemion DialogBlocks, (c) Julian Smart, 2005 Version %.2f, %s")
 wbVERSION_NUMBER, (const wxChar*) date);
        wxLogMessage(msg);
        return false;
    }
    // 检查是否用户希望以调试模式启动
    long debugLevel = 0;
    if (cmdParser.Found(wxT("d"), & debugLevel))
    {
    }
    // 检查是否用户传递了一个工程名
    if (cmdParser.GetParamCount() &gt; 0)
    {
        cmdFilename = cmdParser.GetParam(0);
        // 在windows系统上,如果通过资源管理器打开一个文件的时候,
        // 传递的是短格式的文件名
        // 因此我们可以把它变成长格式文件名
        wxFileName fName(cmdFilename);
        fName.Normalize(wxPATH_NORM_LONG|wxPATH_NORM_DOTS|
                    wxPATH_NORM_TILDE|wxPATH_NORM_ABSOLUTE);
        cmdFilename = fName.GetFullPath();
    }
    ...
    return true;
}
```

使用wxFileName对文件名进行正常化是必要的,因为有时候在以命令行方式启动程序的时候,windows会传递短格式的文件名.

正如我们在前面介绍的那样,在MacOSX上,当打开一个文档的时候不使用命令行参数,而使用直接调用wxApp:: MacOpenFile函数的方法.但是命令行参数的方法确实在多数系统上是被使用的,因此,为了让你开发的程序适用于各种[平台,你还是应该提供命令行参数的支持.

20.6 存储应用程序资源

一个简单的小程序可能只有一个可执行文件.但是,更常见的情形是,你必须使用包括帮助文件,也许还有别的HTML文件和图片文件,以及应用程序自定义的数据文件.这些附带的文件怎么存储合适呢?

减少数据文件的数量

你可以通过一些方法来减少你需要使用的数据文件,以便创建一个更简洁的发行包.首先,对于XPM类型的图片,尽可能在你的代码中使用#include,而不是通过从文件中读取的方法来加载图片.其次,如果你正在使用XRC文件工具,位于wxWidgets发行版的utils/wxrc目录中的wxrc工具能将它变成C++的代码,如下所示:

```
wxrc resources.xrc --verbose --cpp-code --output resources.cpp
```

然后,你就可以调用生成的C++文件中的InitXmlResource函数来初始化这些资源了.

第三种方法是,你可以将所有的数据文件打包成一个zip文件,然后使用我们在前面介绍的流操作和虚拟文件系统的方法来访问它们.你可能还需要用到类wxStandardPaths,它定义在"wx/stdpaths.h"文件中,它的一些静态的成员函数包括GetConfigDir, GetInstallDir,GetDataDir,GetLocalDataDir和GetUserConfigDir等.具体这些函数在各个平台上返回的目录的为止参考wxWidgets手册中的相关描述.

在Mac OSX平台上,你需要创建一个应用程序发布包文件,这个文件用来描述你的应用程序的可执行文件路径,数据文件等.本章稍后部分我们会详细讨论有关的情况.

找到应用程序所在的位置

经常会有wxWidgets的使用者希望能够提供一个函数用来找到应用程序所在的绝对路径,以便可以从同样的路径加载资源文件.不过, wxWidgets并没有提供这样的函数,这部分是因为要在不同的平台上实现这些函数是有一定难度的,可能会返回不可靠的路径,也是部分出于鼓励开发者最好把数据文件放在系统标准的数据文件夹中(尤其是在linux系统上)的原因. 然而,将所有应用程序相关的文件都放在一个路径下也是可以理解的,因此,在随书光盘的examples/chap20/findapppath目录中,你可以找到一个函数wxFindAppPath的代码,用来实现这个功能,它的声明部分如下:

```
// 返回当前正在运行的可执行文件的绝对路径
wxString wxFindAppPath(const wxString& argv0, const wxString& cwd,
                                          const wxString& appVariableName = wxEmptyString,
                                          const wxString& appName = wxEmptyString);
```

argv0的值等于wxApp::argv[0],在某些平台上,它代表了当前执行文件的完整路径.

cwd是当前工作目录(你可以通过调用wxGetCwd函数得到),在某些平台上我们需要根据这个参数作出一些判断.

appVariableName是相关环境变量的值,比如环境变量MYAPPDIR,这些变量可以被在程序外部设置用来指明应用程序查找位置.

appName是你在发行包中指明的前缀,函数可能需要使用它来检查位于发行包中的某些路径.比如,DialogBlocks程序的这个参数是DialogBlocks,因此在Mac OsX系统上,这个函数会在<currentdir>/DialogBlocks.app/Content/MacOS中寻找可执行文件的全路径.

下面是这个函数的使用方法举例:

```
bool MyApp::OnInit()
{
    wxString currentDir = wxGetCwd();
    m_appDir = wxFindAppPath(argv[0], currentDir, wxT("MYAPPDIR"),
                             wxT("MyApp"));
    ...
    return true;
}
```

在Windows平台和Mac OSX平台上,这个函数返回的路径都是可以信赖的,然而在Unix平台上,只有应用程序是从其所在的目录被启动的时候,返回的路径才是可以信赖的.或者如果你正确设置了MYAPPDIR这个环境变量,返回的路径也是可以信赖的.因此,为了让返回的值更正确,有些安装程序选择另外安装一个启动脚本,这个脚本会首先设置MYAPPDIR环境变量,然后再启动应用程序.你可以选择提示用户是否安装这个脚本,或者你可以直接把你的程序安装在标准的路径上,比如 /usr/local/bin/

20.7 调用别的应用程序

有时候你需要从你的应用程序中启动别的应用程序,可能是一个浏览器或者是你自己写的另外一个程序.wxExecute函数是一个功能很强大的函数,它的功能包括: 带参数或者不带参数调用别的程序,同步或者异步执行程序,搜集别的程序的输出,以及重定向别的程序的输入和输出以便实现和当前程序的交互.

启动一个应用程序

下面是wxExecute函数的一个简单的例子:

```
// 异步执行程序(默认行为),函数将会立即返回.
wxExecute(wxT("c:\\windows\\notepad.exe"));
// 同步执行程序,函数在Notepad程序退出以后才会返回.
wxExecute(wxT("c:\\windows\\notepad.exe c:\\temp\\temp.txt"),
          wxEXEC_SYNC);
```

注意一般来说你可以将参数和可执行文件用引号括起来,这在路径中包含空格的时候是很有用的.

打开文档

如果你启动一个外部程序的目的是打开一个文档,在Windows或者Linux平台上,你可以使用wxMimeTypesManager 类.你可以使用它来获得打开某种类型的文档所需要执行的程序的路径,然后使用它来构造wxExecute函数的参数,比如,如果你想打开一个HTML文件,你可以使用下面的方法:

```
wxString url = wxT("c:\\home\\index.html");
bool ok = false;
wxFileType *ft = wxTheMimeTypesManager->
                 GetFileTypeFromExtension(wxT("html"));
if ( ft )
{
     wxString cmd;
      ok = ft->GetOpenCommand(&cmd,
                         wxFileType::MessageParameters(url, wxEmptyString));
      delete ft;
      if (ok)
      {
           ok = (wxExecute(cmd, wxEXEC_ASYNC) != 0);
       }
}
```

不幸的是,这种方法不适用于Mac OSX平台,因为Mac OSX平台使用完全不同的文档打开机制.对于任何别的文件类型,最好使用系统提供的Finder程序来打开,而对于HTML文件,你可以直接使用系统函数 ICLaunchURL.wxExecute有时候并不是最好的选择,在windows平台上,如果要打开HTML文件,你可以直接使用 ShellExecute函数会更有效率.即使在Unix平台上,你可能也要作好指定的程序不存在的准备,如果它确实不存在,你可以考虑使用别的程序比如 htmlview.

为了避免上述的这些问题,我们在随书光盘的examples/chap20/launch目录中,实现了一些函数,比如: wxLaunchFile,wxViewHTMLFile,wxViewPDFFile,wxPlaySoundFile,它们的功能一目了然,并且它们可以同时支持Windows,Linux和Mac OsX平台.

wxLaunchFile是一个普通意义上的文本打开函数.参数包括一个文档文件名或者一个可执行文件名附带可选的参数,以及一个可选的错误消息字符串,这个字符串在执行失败的时候显示给用户.如果当前正在打开的文档是HTML类型的文档,wxLaunchFile函数将调用 wxViewHTMLFile函数.在Mac OsX平台上,这个函数将使用Finder打开文档,而在别的平台上则使用wxMimeTypesManager.注意在Mac OSX平台上,有时候文档会在非活动的窗口上打开,这时候你可以通过osascript这个命令行工具来将它提到前台,如下所示(比如):

```
wxExecute(wxT("osascript -e \"tell application \\\"AcmeApp\\\"\" -e
\"activate\" -e \"end tell\""));
```

在Linux平台上, wxViewHTMLFile, wxViewPDFFile和wxPlaySoundFile都包含fallbacks机制以便在相应的可执行文件不存在的时候使用.你可以按照自己的需要调整相应的fallbacks设置. wxPlaySoundFile是用来使用外部程序播放那些大型的声音文件的,如果只是播放一个很小的声音文件,你可以直接使用wxSound.

重定向进程的输入和输出

有时候,你希望捕获另外一个进程的输入和输出,以便你或者你的用户可以控制那个进程.比起重头写实现某个功能的代码来说,这样作显然可以给你减少不少的工作量.而wxExecute可以帮助实现捕获和控制那些控制台程序的输入和输出.

要实现这个功能,你需要在调用wxExecute函数的时候传递一个wxProcess的实例,这个实例的OnTerminate函数将在进程结束的时候被调用,这个实例可以用来捕获进程的输出或者控制进程的输入.

在wxWidgets自带的samples/exec目录中,你可以找到各种各样使用wxExecute的例子,我们也提供了另外一个例子,它将GDB集成进自己的程序中去,你可以参考examples/chap20/pipedprocess中的代码.我们没有提供用于工具条的那些小图片以及整个可编译的代码,如果提供了这些,它将可以支持包括windows,linux和Mac OSX在内的各种平台,只要那些平台上安装了GDB.

debugger.h和debugger.cpp文件实现了一个管道化的进程和一个窗口,这个窗口包含一个工具条和一个文本框,用来显示GDB的输出和从用户那里获取输入并且把它发送给GDB.

textctrlex.h和textctrlex.cpp则实现了一个派生自wxStyledTextCtrl的控件,包括一些和wxTextCtrl兼容的函数和标准事件处理函数比如复制,剪切,粘贴,重做和撤消等.

processapp.h和processapp.cpp实现了一个应用程序类,这个类可以在空闲的时候处理来自多个进程的输入.

GDB是通过下面的语句启动的:

```
DebuggerProcess *process = new DebuggerProcess (this);
m_pid = wxExecute(cmd, wxEXEC_ASYNC, process);
```

可以使用下面的代码杀死这个进程:

```
wxKill(m_pid, wxSIGKILL, NULL, wxKILL_CHILDREN);
```

要给调试器发送一个命令,将会设置一个内部的变量以便通知应用程序在空闲的时候处理这个输入.

```
// 给调试器发送一个命令
bool DebuggerWindow::SendDebugCommand(const wxString& cmd,
                                             bool needEcho)
{
      if (m_process && m_process->GetOutputStream())
      {
           wxString c = cmd;
           c += wxT("\n");
           if (needEcho)
                AddLine(cmd);
           // 这个函数只是简单的对m_input变量赋值
           // OnIdle函数中的HasInput函数将检查这个变量.
           m_process->SendInput(c);
           return true;
      }
      return false;
}
```

HasInput函数被应用程序在其空闲时间周期性的调用,它的责任是发送用户输入的命令到进程并且从进程读取来自标准输出和标准错误的输出:

```
bool DebuggerProcess::HasInput()
{
       bool hasInput = false;
       static wxChar buffer[4096];
       if ( !m_input.IsEmpty() )
       {
            wxTextOutputStream os(*GetOutputStream());
             os.WriteString(m_input);
             m_input.Empty();
             hasInput = true;
        }
        if ( IsErrorAvailable() )
        {
              buffer[GetErrorStream()->Read(buffer, WXSIZEOF(buffer) -
1).LastRead()] = _T('\0');
               wxString msg(buffer);
                m_debugWindow->ReadDebuggerOutput(msg, true);
                hasInput = true;
         }
         if ( IsInputAvailable() )
         {
                 buffer[GetInputStream()->Read(buffer, WXSIZEOF(buffer) -
1).LastRead()] = _T('\0');
                wxString msg(buffer);
                m_debugWindow->ReadDebuggerOutput(buffer, false);
                hasInput = true;
         }
         return hasInput;
}
```

注意上面这个例子和wxWidgets自带的exec例子的一个关键的不同在于,exec例子每次从进程读取一行,如果进程的输出没有带换行符,将导致应用程序被阻塞.而在我们的例子中,使用了一个缓冲区来保存尽可能多的输入,这是一种更安全的作法.

ProcessApp类可以直接被用作你的应用程序的基类,或者你可以拷贝它的成员函数到你的应用程序类中去.它维护了一个进程列表,进程可以通过RegisterProcess和UnregisterProcess函数登记和注销, 进程输入和输出的处理在系统空闲时间完成.如下所示:

```
// 任何缓存的输入都在系统空闲时处理
bool ProcessApp::HandleProcessInput()
{
    if (!HasProcesses())
         return false;
    bool hasInput = false;
    wxNode* node = m_processes.GetFirst();
    while (node)
    {
        PipedProcess* process = wxDynamicCast(node->GetData(), PipedProcess);
        if (process && process->HasInput())
            hasInput = true;
        node = node->GetNext();
    }
    return hasInput;
}
void ProcessApp::OnIdle(wxIdleEvent& event)
{
    if (HandleProcessInput())
        event.RequestMore();
    event.Skip();
}
```

20.8 管理应用程序设置

大多数的应用程序都会给用户一些选项,以便用户自己决定一些应用程序的行为,比如是否显示每日提示,文本应用什么字体,或者是否显示启动画面等.而程序员需要作的决定是怎样保存和显示这些配置数据.关于怎样存储,通常我们需要使用wxConfig家族的类,这些类让你可以直接处理类型化的配置数据.至于如何显示,则是非常灵活的,我们将简短的介绍一些可能的选项.

保存配置数据

所有wxWidgets提供的用于处理配置数据的类都是wxConfigBase的派生类,因此你可以在这个基类的手册中找到相关的使用方法.而wxConfig则被定义为各个平台上推荐使用的用于处理配置数据的类:在windows平台,它被定义为wxRegConfig(这个类使用 windows的注册表),在所有别的平台上它被定义为wxFileConfig(它使用文本文件).另外还有wxIniConfig类,它使用一个Windows 3.1风格的.ini配置文件,不过这个类很少被使用到.而wxFileConfig则可以支持各个平台.

wxConfig类提供了各种Read和Write函数的重载函数,用来直接读写各种数据类型,包括wxString, long, double和bool类型等. 配置文件中的每个项目都需要提供一个路径,这个路径由"/"分割并且最后必须是一个名称,比如"/General/UseTooltips".你可以使用wxConfig::SetPath函数设置一个当前路径,这样的话,在后续的读写中,如果没有指定绝对路径(以"/"开头),所有的路径都被认为是相对于这个路径的路径. 使用路径的目的是为了对配置项进行分组.

wxConfig的构造函数需要使用应用程序名和供应商名称,这两个名称用来决定配置项的位置,比如:

```
#include "wx/config.h"
wxConfig config(wxT("MyApp"), wxT("Acme"));
```

wxRegConfig将会从应用程序名和供应商名称构造一个注册表项,比如前面的例子中将会导致注册表项 HKEY_CURRENT_USER/Software/Acme/MyApp被创建.而如果是Unix系统上的wxFileConfig类,配置文件默认被保存在文件~/.MyApp中. 而在Mac OSX上,则保存在/Library/Preferences/MyApp/Preferences中.这些缺省位置可以通过给wxConfig传递第三个参数来改变:

下面是一些wxConfig的用法:

```
// 读取
wxString str;
if (config.Read(wxT("General/DataPath"), & str))
{
    ...
}
bool useToolTips = false;
config.Read(wxT("General/ToolTips"), & useToolTips));
long usageCount = 0;
config.Read(wxT("General/Usage"), & usageCount));
// 写入
config.Write(wxT("General/DataPath"), str))
config.Write(wxT("General/ToolTips"), useToolTips));
config.Write(wxT("General/Usage"), usageCount));
```

其它一些可以使用的操作包括比例组和选项条目,查询某个组或者某个选项是否存在,删除一个条目或者一个组等.

你可以临时使用wxConfig来读取一些存放在某个地方的数据,你也可以创建一个wxConfig的实例并且在应用程序的整个生命周期维持它.wxWidgets也有一个称为默认wxConfig对象的机制,这个默认的对象可以通过wxConfig::Set函数设置.如果设置了这个默认对象,一些wxWidgets的内部实现将会使用这个对象,比如wxFontMapper类或者平台通用的wxFileDialog实现.

编辑选项

如果你只有很少的选项,那么普通的对话框也许就足够了.但是有时候,选项有很多,并且非常复杂,这时候,你可能需要很多对话框或者面板, 这种情况下最通常的作法是使用包含一个wxNotebook的模式对话框,这个对话框的底部应该有OK,Cancel或者Help按钮.其中Help按钮的处理函数将会查询当前正在显示的页面并显示一个相应的帮助文件主题.wxWidgets提供了一个叫做wxPropertySheetDialog的对话框来处理这种情形.wxWidgets自带的samples/dialogs例子中演示了它的使用方法.在Pocket PC上,这个对话框里的notebook控件将显示成屏幕底部标准的属性页面.

你也可以使用wxListbook和wxChoicebook来代替wxNotebook,它们是控制多页控件的又一个选择.尤其是 wxListbook,它的API和wxNotebook几乎相同,但是它使用wxListCtrl而不是TAB来控制页标签,因此你可以使用图标和标签来代替TAB按钮.这在你拥有很多页面的时候也很有用.尤其是在Mac OSX平台上,这个平台的wxNotebook控件不能够自己滚动标签按钮,因此标签按钮的数目受限于wxNotebook的宽度和标签的宽度.另外你也可以下载第三方的awxOutbarDialog控件,它实现了一个类似Outlook外观的那种多页控件,使用图标来在页面间切换.

你也可以创建自定义的分页管理对话框,比如你可以使用wxtreeCtrl控件,这可以让你的各个页面保持一种树状的继承关系.要实现这个自定义控件,你可以维护一组面板列表,每一个都绑定一个名字.当用户点击树状控件上的某个子项的时候,隐藏当前正在显示的面板,而显示树状控件子项对应的面板.另外一个方案是使用Jorgen Bodde制作的wxTreeMultiCtrl控件,这个控件实现了上面所介绍的内容,因此你可以以更直观的方法使用树状分页控件,而不比自己处理每个单独的页面.

你也可以考虑使用一个属性编辑框:这是一个拥有一系列子项,每个子项左边拥有一个文本标签,右边拥有一个编辑框.这个控件的好处在于增加和删除设置是非常容易的,并且不影响界面的布局.不好的地方在于,如果你要编辑多行文本或者编辑一个列表就比较困难,尽管你可以拦截子项的双击事件,使用定制的对话框来显示其内容. 你可以实现自己的属性编辑框,或者你可以考虑使用wxGrid.,或者使用第三方的属性编辑控件比如Jaakko Salli的wxPropertyGrid.有些应用程序混合使用了对话框和属性列表,比如DialogBlocks设置对话框的配置页面.

你最好不要使用带有滚动条的面板或者对话框来避免配置项控件超出范围之外,因为这会是人感觉迷惑并且也是很丑陋的.

你应该考虑将你应用程序的所有设置保存在一个统一的类中,并且为这个类实现一个拷贝构造函数,一个等于操作以及一个赋值操作.通过这种方法,你可以很容易的创建一个所有配置项的副本,将其传递给你的配置对话框,并且仅仅在用户点击了配置对话框上的OK按钮的时候,才将修改的数据保存回你的全局配置中.

如果你没有单独的保存各个配置项,你需要给你的用户提供一种直接修改配置的方法.参考光盘中的 examples/chap20/valconfig例子.其中包含了一个类wxConfigValidator,这个类可以用来作为普通控件的验证器, 它的参数包括配置项路径,配置项类型以及一个指向wxConfig对象的指针.其中值类型可以是wxVAL_BOOL, wxVAL_STRING或者wxVAL_LONG.如下所示:

```
void MyDialog::SetValidators(wxConfig* config)
{
    FindWindow( ID_LOAD_LAST_DOCUMENT )->SetValidator(
        wxConfigValidator(wxT("LoadLastDoc"), wxVAL_BOOL, config));
    FindWindow( ID_LAST_DOCUMENT )->SetValidator(
        wxConfigValidator(wxT("LastDoc"), wxVAL_STRING, config));
    FindWindow( ID_MAX_DOCUMENTS)->SetValidator(
        wxConfigValidator(wxT("MaxDocs"), wxVAL_LONG, config));
}
```

第9章中介绍了更多关于验证器的知识.本节提到的那些第三方控件可以在附录E,"wxWidgets的三方控件"中找到.

20.9 应用程序安装

如果你的应用程序可以很顺利的安装到用户的电脑上,这无疑在用户开始使用你的程序之前就给用户一个很不错的第一印象.在这一节里,我们将依次介绍在Windows,Linux和OsX平台上怎样制作安装程序,其中涉及到的一些第三方工具可以在附录E中找到.

在Windows系统上安装你的程序

在windows平台上,我们尤其需要一个安装程序,这不只是因为用户期待这样,而且安装程序还需要作一些类似文件类型绑定和创建快捷方式这样的动作.

不够这实在和wxWidgets所关注的邻域差别太大,因此wxWidgets并不准备自己提供这样的工具.一些另外的工具可以用来创建安装程序,比如 NSIS和InstallShield;另外一个广受好评的软件是Inno Setup,它是一个非常强大的,免费的安装程序制作工具,它可以通过脚本来定制安装文件选项,通过Pascal语言来对其现有功能进行扩展.它的网站上也列举了一些用来创建安装脚本的图形界面工具.

如果你需要很频繁的发布新的版本,你可能想要通过一个脚本自动创建安装程序.随书光盘的examples/chap20/install目录中提供了一个用于创建这样的脚本的例子,你可以按你的需要进行更改.因为它们是Unix风格的 Shell脚本,需要你有MingW或者MSys的环境,这些环境也有在随书光盘中提供.你需要提供的包括一个放置文件的目录,makeinno.sh脚本将会创建Inno Setup的脚本中枚举子目录和文件的部分.而安装脚本的头和尾部那些需要你自己按照自己软件的情况提供的部分将不会被自动创建.你可以使用下面的命令来创建安装文件:

```
sh makeinno.sh c:/temp/imagedir innotop.txt innobott.txt myapps.iss
```

这将会基于文件夹c:/temp/imagedir中的文件创建Inno Setup的脚本文件myapp.iss.

你可以调整makesetup.sh脚本来创建你需要的安装程序,这个文件首先将需要的文件拷贝到一个"images"文件夹(就是前面 makeinno.sh脚本需要的那个文件夹),然后创建setup.exe.这个脚本使用了定义在setup.var中的变量.你可以按照你自己的情况增加新的功能,比如编译你的应用程序或者使用Curl工具拷贝文件到你的FTP站点等.

当你发布应用程序的时候,别忘了增加一个WindowsXp的"manifest"文件.这个文件是一个Xml格式的文件,用来告诉 WindowsXp应该给这个程序应用什么风格.你可以通过在你的程序的资源文件(.rc)中增加wxWidgets标准资源文件的方式来增加这个文件. 如下所示:

```
aardvarkpro ICON aardvarkpro.ico
#include "wx/msw/wx.rc"
```

这将包含一个标准的manifest文件,如果你希望使用自己定义的manifest文件,在包含wx.rc之前,你需要定义wxUSE_NO_MANIFEST宏,然后再指定你自己的manifest文件,如下所示:

```
aardvarkpro ICON aardvarkpro.ico
#define wxUSE_NO_MANIFEST 1
#include "wx/msw/wx.rc"
1 24 "aardvark.manifest"
```

你也可以直接将manifest文件放在你的应用程序目录中,详情可参考wxWidgets发行版自带的docs/msw/winxp.txt文件.

在Linux系统上制作安装程序

在Linux系统,你可以选用图形界面的安装程序,定制的shell脚本或者某个特定发行版的软件包,比如RPM格式(基于Red Hat发行版)和Debian发布包(基于Debian发行版),你甚至可以直接将所有需要的文件压缩成一个包含路径的压缩文件(.tar.gz或者. tar,bz2),安装的时候只需要保持路径解压这个文件就可以了.

Linux环境下的图形界面安装程序包括Loki Setup(免费),Zero G公司的InstallAnywhere和InstallShield等.

基于GTK+的wxWidgets图形界面应用程序是桌面不可感知的:它不依赖于GNOME或者KDE,因此无论在哪种桌面环境下它都可以运行.大多数KDE桌面的发行版都会包括GTK+的库文件.然而,因为它们使用不同的桌面风格,GTK+程序在KDE桌面上看上去可能会有些不适应,某些控件可能超出边界,这种情况下你可以建议你的用户安装一个KDE下的GTK风格的皮肤,比如GTK-Qt(不过,在你把它介绍给你的用户之前,最好自己先测试一下).

你可能会希望在桌面上安装一个图标,以便你的用户可以直接使用它来启动你的程序.要在KDE桌面环境中增加一个图标,你需要拷贝一个合适的APP.desktop文件到PREFIX/share/applications文件夹,其中APP代表你的应用程序,PREFIX则通常代表/usr,/usr/local或者其它定义在KDEDIR环境变量中的路径.下面演示了一个叫做Acme的Desktop文件,其中架设Acme被安装在/opt/Acme中.

```
[Desktop Entry]
BinaryPattern=Acme;
MimeType=
Name=Acme
Exec=/opt/Acme/acme
Icon=/opt/Acme/acme32x32.png
Type=Application
Terminal=0
```

而要在GNOME桌面上增加一个图标,语法和KDE中相似不过放置的位置应该是~/.gnome-desktop(只对单个用户有效). 更多关于GNOME和KDE桌面文件的定义可以在下面的网址看到:http: //www.freedesktop.org/wiki/Standards_2fdesktop_2dentry_2dspec.

如何制作RPM包的信息可以在http://www.rpm.org找到,那里还包含一个免费的在线电子书.而创建Debian包的信息可以在http://www.debian.org找到.这俩中方法创建的安装包可以允许系统进行依赖性检查,也使得用户可以很容易的浏览软件包的内容和安装软件包.如果需要创建RPM,.deb或者其它格式发行包的软件,可以试一下EPM.

关于使用shell脚本创建Linux的安装文件的方法,随书光盘的examples/chap20/install目录中包含了一个用来安装Acme的示例文件installacme.这个脚本作的事情包括安装整个程序并且创建一个叫做acme脚本,这个脚本在运行实际的可执行文件之前会先设置当前位置环境变量.这样作的好处在于你既不需要将软件所在的目录的路径增加到PATH环境变量中,也不需要将可执行文件直接拷贝到系统路径下就可以执行.所有的数据文件保持在可执行文件所在的目录中.这使得卸载软件变得容易.你当然也可以选择让安装脚本将数据放置到Linux的标准数据目录中.

在examples/chap20/install目录中还包含一个脚本叫做maketarball.sh,它演示了怎样创建一个用户发行的tar格式的压缩包,installacme脚本将包含在这个压缩包内以及另外一个包含所有数据文件的压缩包.你可以修改maketarball.sh以满足你自己的需要.

Linux环境上的动态链接库的问题

因为Linux系统上并没有标准的GUI库,各个发行版按照自己的喜好来添加它喜欢的库和程序,因此你可能会发现在某些系统上,你的应用程序不能运行,提示的原因是无法找到动态链接库.因此,静态链接所有需要的库文件实在是一个很诱人的想法.但是这样又会导致别的一些问题.虽然你不应该静态链接GTK+的那些库文件,但是静态链接wxWidgets的库是可行的,你可以在运行configure脚本的时候选择开关--disable- shared以达到这个目的.你也可以考虑将wxWdigets提供的那些动态链接库以及所需要的GTK+相关的库和你的应用程序打包在一起发布.

另外就是不要在太老的linux发行版(太老的那些动态链接库在新版上已经不提供了)或者太新的Linux发行版(需要一些老的发行版上还没有的库)上编译你的软件.同时考虑在给你的链接器增加-lsupc++ 选项,以便你的程序可以静态链接一些基本的C++的库,而不是需要完全的依赖动态链接库,这样作可能会解决一些潜在的问题(不过,请注意静态链接GPL库时候的版权问题).

最后,如果你想要针对各个发行版发布不同的软件包,如果你不想老是重新启动电脑以切换到不同的Linux,你可以考虑使用一个工具比如伟大的VMware,它可以让你同时在你的机器上运行多个linux的发行版.

在Mac OSX上安装程序

在Mac OSX上,你真正需要作的就是确认你的软件的目录结构是正确的,然后将它作成一个合适的磁盘镜像文件,我们将简单的介绍一下.Mac OSX上没有安装程序制作工具这种东西,你只需要将你的文件夹拖到磁盘的合适的位置就可以了.

下面我们大概来介绍一下Mac的软件包结构,你可以在苹果公司的网站http://developer.apple.com/documentation/MacOSX/Conceptual/SystemOverview/Bundles/chapter_4_section_3.html 找到更多的信息.

一个软件包包含一个标准的目录结构和一个Info.plist文件,这个文件用来描述软件包的某些属性.

一个小的软件包结构如下所示:

```
DialogBlocks.app/                    ; top-level directory
    Contents/
        Info.plist                   ; the property list file
        MacOS/
            DialogBlocks             ; the executable
        Resources/
            dialogblocks-app.icns    ; the app icon
            dialogblocks-doc.icns    ; the document icon(s)
```

下面是一个用于DialogBlocks软禁的Info.plist文件的例子:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
<dict>
      <key>CFBundleInfoDictionaryVersion</key>
      <string>6.0</string>
      <key>CFBundleIdentifier</key>
      <string>uk.co.anthemion.dialogblocks</string>
      <key>CFBundleDevelopmentRegion</key>
      <string>English</string>
      <key>CFBundleDocumentTypes</key>
      <array>
            <dict>
                  <key>CFBundleTypeExtensions</key>
                  <array>
                        <string>pjd</string>
                  </array>
                  <key>CFBundleTypeIconFile</key>
                  <string>dialogblocks-doc.icns</string>
                  <key>CFBundleTypeName</key>
                  <string>pjdfile</string>
                  <key>CFBundleTypeRole</key>
                  <string>Editor</string>
            </dict>
      </array>
      <key>CFBundleExecutable</key>
      <string>DialogBlocks</string>
      <key>CFBundleIconFile</key>
      <string>dialogblocks-app.icns</string>
      <key>CFBundleName</key>
      <string>DialogBlocks</string>
      <key>CFBundlePackageType</key>
      <string>APPL</string>
      <key>CFBundleSignature</key>
      <string>PJDA</string>
      <key>CFBundleVersion</key>
      <string>1.50</string>
      <key>CFBundleShortVersionString</key>
      <string>1.50</string>
      <key>CFBundleGetInfoString</key>
      <string>DialogBlocks version 1.50, (c) 2004 Anthemion Software Ltd.</string>
      <key>CFBundleLongVersionString</key>
      <string>DialogBlocks version 1.50, (c) 2004 Anthemion Software Ltd.</string>
      <key>NSHumanReadableCopyright</key>
      <string>Copyright 2004 Anthemion Software Ltd.</string>
      <key>LSRequiresCarbon</key>
        <true/>
      <key>CSResourcesFileMapped</key>
      <true/>
</dict>
</plist>
```

程序使用的图标和支持的文档类型是通过CFBundleIconFile和CFBundleTypeIconFile属性指定的.正如我们在第10章,"使用图片编程"中介绍的那样,如果你主要实在Windows或Linux下编程,你可以创建各种不同大小的 (16x16,32x32,48x48和128x128)的图标文件.将其保存为透明的PNG文件,然后拷贝到Mac平台上,在Finder中打开这些文件,将其拷贝和粘贴到苹果公司的图标编辑器中,然后就可以另存为icns文件了.

前面我们介绍过的maketarball.sh脚本也可以用来创建Mac OSX上的磁盘镜像文件.比如AcmeApp-1.50.dmg.它将已经准备好的AcmeApp.app包中的目录结构拷贝到新的目录结构,然后拷贝用于Mac OSX的可执行文件和数据文件,然后再使用下面的代码创建一个可以直接用于Internet安装的磁盘镜像文件:

```
echo Making a disk image...
hdiutil create AcmeApp-$VERSION.dmg -volname AcmeApp-$VERSION -type UDIF -megabytes 50 -f
echo Mounting the disk image...
MYDEV=`hdiutil attach AcmeApp-$VERSION.dmg | tail -n 1 | awk '{print $1'}`
echo Device is $MYDEV
echo Copying AcmeApp to the disk image...
ditto --rsrc AcmeApp-$VERSION /Volumes/AcmeApp-$VERSION/AcmeApp-$VERSION
echo Unmounting the disk image...
hdiutil detach $MYDEV
echo Compressing the disk image...
hdiutil convert AcmeApp-$VERSION.dmg -format UDZO -o AcmeApp-$VERSION-compressed.dmg
echo Internet enabling the disk image...
hdiutil internet-enable AcmeApp-$VERSION-compressed.dmg
echo Renaming compressed image...
rm -f AcmeApp-$VERSION.dmg
mv AcmeApp-$VERSION-compressed.dmg AcmeApp-$VERSION.dmg
```

之后,新创建的磁盘镜像文件就可以拷贝到你的FTP站点或者CD-ROM站点.当你的用户在一个浏览器中点击这个文件的时候,文件就会被自动下载,解包,加载成一个虚拟的磁盘,所有这些过程都不需要用户的干预,然后就等着用户把整个软件包拖拽到磁盘的合适的位置,就可以完成软件的安装了.

20.10 遵循用户界面设计规范

学习各个平台的界面设计规范是一件很值得一做的事情.虽然它们中的绝大多数差异都已经被wxWidgets自动屏蔽了,不过还是有一些细节是无法自动解决的.比如按钮的布局风格在不同的平台上是不一样的.苹果的Mac OSX操作系统上按钮的顺序和间隔的要求是非常严格的.下面只是我们认为值得特别说明的一些方面,包括一些平台相关的规则和一些一般的规则.另外你也可以通过多操作各个平台上的经典的程序,并观察他们外观的不同来帮助你设计你自己的程序在这些平台上的外观.

标准按钮

在windows和Linux平台上,按钮可以被整体居中或者右对齐,通常的顺序是OK,Cancel和Help.而在Mac OSX上,帮助按钮(如果使用wxID_HELP会自动显示一个问号标记)通常应该是左对齐的,其它的按钮则是右对齐的,并且最右边的那个是默认按钮,也就是说是:?号,空格,Cancel,OK这样的顺序.

尽可能使用wxWidgets提供的标准按钮标识符(比如wxID_OK, wxID_CLOSE, wxID_APPLY等),因为在某些平台上(尤其是wxGTK平台上),这些标准标识符会被自动添加一些合适的图形.

参考第7章"使用布局控件进行窗口布局"中的"平台相关布局"小节了解wxStdDialogButtonSizer类的使用方法,这个类能够对标准按钮进行平台相关的布局.

菜单

避免出现空的菜单条.小心的给各个菜单添加有意义的标签,并且使用"&"符号引导的通常是标签的第一个字符的加速键(比如 &File)和快捷键(比如Ctrl+O).常用的那些菜单项命令应该尽可能的提供,比如拷贝,粘贴,撤消等.不要有太长或者太短的菜单项.通常 9到10个菜单项是一个比较合理的最大值.如果确实有很多选项需要配置,可以考虑使用一个菜单项弹出一个对话框进行这些设置.

和按钮的使用一样,尽可能使用wxWidgets提供的标准的标识符,尤其是wxID_HELP,wxID_PREFERENCES 等,在Mac OSX平台上,wxID_HELP菜单将被移动到应用系统菜单中去,你应该注意这个问题,以便产生空的菜单条或者连续两个菜单分割条.

图标

你工具栏,frame窗口和别的界面元素上的图标可以给你的应用程序一个很好的观感.忽略这一点可以让你的应用程序的界面效果大打折扣. 尤其是在MAc OSX平台上,这个平台对于美学的要求是很高的.你应该尽量给每一个项目创建自己的图标,或者,一个更简单的作法,直接购买别人设计好的图标,然后将那些非标准的图标按照统一的风格进行设计.在图标上的付出将会获得等价的回报,你的应用程序将会因为使用了这些图标而增光添彩,也会给用户留下很强烈的印象.你也可以在网上找到一些图标,比如,遵循L-GPL协议发布的Ximian图标集: http://www.novell.com/coolsolutions/feature/1637.html .

字体和颜色

不要在你的对话框上使用很多中字体和颜色,这除了导致你的界面看上去花里胡哨以外,还使得wxWidgets很难去进行针对各个平台的一些外观调整,以便给出你的应用程序以本地观感.不过,这并不防碍你给你的用户增加更改默认字体的选项,以便他们可以改变那些包含很多纹理信息的对话框的外观,比如用作报告的对话框.对于颜色的使用要遵循那个平台的规范.对于wxWidgets提供的对话框,wxWidgets可以自己作一些平台适应工作, 但是对于你自定义的对话框,有些则需要你自己去注意这个问题.

应用程序中止时的行为

在大多数平台上,没有使用MDI界面或者将类似界面的基于文档的应用程序将在每个frame窗口中显示一个文档.当最后一个文档被关闭的时候应用程序就将退出.但是在Mac OSX平台上,正如我们在第19章"使用文档和视图框架"中介绍的那样,用户并不期待这时候整个应用程序退出,应用程序应该还有一个菜单显示在系统菜单条上,以便用户可以通过它打开或者创建新的文档或者关闭应用程序.这可以通过创建一个不可见的frame主窗口的方法来实现,可能需要你增加一点点平台相关的代码.

在嵌入式开发系统中(比如Pocket PC),应用程序在主窗口被关闭的时候仍然停留在内存中,用户通常没有办法让他们退出.你可以选择是否遵守这个规则还是允许用户退出应用程序以便给别的程序腾出内存.在Pocket PC上,wxWidgets也会设置标准的快捷键Ctrl+Q用来退出应用程序,这个快捷键的默认处理动作是发送wxID_EXIT命令事件.

进一步阅读

下面列出了wxWidgets支持的各个主要平台上的用户界面设计规范,以及一些一般性UI设计建议的书籍:

- 苹果用户界面设计规范: http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/index.html
- Mac OSX和Windows用户界面的关键差异: http://developer.apple.com/ue/switch/windows.html
- 微软官方用户界面设计规范: http://msdn.microsoft.com/library/default.asp?url=/library/enus/dnwue/html/welcome.asp
- GNOME用户界面设计规范: http://developer.gnome.org/projects/gup/hig
- GUI Bloopers: 软件开发和Web设计中要做和不要作的事, 作者:Jeff Johnson (Academic Press). ISBN 1-55860-582-7
- 程序员用户界面设计, 作者: Joel Spolsky (Apress). ISBN 1-893115-94-1
- 软件可用性: 以可用性为核心进行软件设计和建模(A Practical Guide to the Models and Methods of Usage-Centered Design), 作者:Larry L. Constantine and Lucy A.D. Lockwood (ACM Press). ISBN 0-201-92478-1

20.11 全书小结

本章我们介绍了完善你的程序相关的各种主题,演示了一些弥补wxWidgets不足之处的代码.最后我们介绍了一些用户界面设计的有益提示,介绍了一些更进一步介绍UI规范的书籍.

我们希望通过这些书籍的阅读,能够让你更加认同我们的工作,更加认同wxWidgets,它是一个非常强大的工具集,它可以给予你的东西包括:

- 你的应用程序将拥有本地观感
- 大量的类控件,包括各种简单和复杂的窗口控件,轻量级的HTML支持,向导,联机帮助,多线程,进程间通信,流及虚拟文件系统等等,将让你开发出更加稳健的产品级的程序,并且让你享受开发的过程.
- 当然,在各个平台上使用同样的代码也将为你节省很多钱.
- 你可以很容易的将你的代码移植到别的你正打算移植的平台,比如Pocket PC和Mac OS X,以便为它赢得更大的市场.
- 通过使用快速开发工具比如DialogBlocks,以及强大的布局控件机制,你可以很快的创建出复杂而优雅的,可伸缩的并且是可移植的对话框和窗口.
- wxWidgets是开放源代码的,你可以更改它的代码或者理解它到底是怎样工作的.
- 你将从wxWidgets庞大的社区支持中受益,你的问题将很快被答复,你还可以使用很多第三方的控件和工具包(参考附录E)

我们非常希望你能够享受阅读这本书的过程,并且在浏览了光盘中的例子和工具之后,愿意马上开始将你学到的这些知识应用到你的跨平台程序中去.祝你好运,我们期待很快能够在wxWidgets的邮件列表或者论坛上看到你的身影.